Java 8 Features

05 July 2024 12:06 PM

1. Java Lambda Expressions

- The Lambda expression is used to provide the implementation of an interface which has functional interface.
- It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation.
- Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface:

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Q. Why use Lambda Expression

1.To provide the implementation of Functional interface.

2.Less coding.

Java Lambda Expression Syntax

```
(argument-list) -> {body}
```

❖ Java lambda expression is consisted of three components.

1) **Argument-list:** It can be empty or non-empty as well.
2) **Arrow-token:** It is used to link arguments-list and body of expression.
3) **Body:** It contains expressions and statements for lambda expression.

❖ A lambda expression can have zero or any number of arguments.

◇ Java Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
```

```
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Output:

I have nothing to say.

◇ Java Lambda Expression Example: Single Parameter

```
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

Output:

Hello, Sonoo Hello, Sonoo

◇ Java Lambda Expression Example: Multiple Parameters

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
```

```
        System.out.println(ad2.add(100,200));
    }
}
```

Output:

30

300

2. Java Functional Interfaces

- An Interface that contains exactly one abstract method is known as functional interface.
- It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.
- Functional Interface is also known as Single Abstract Method Interfaces or SAM Interface.

```
1. @FunctionalInterface
2. interface sayable{
3.     void say(String msg);
4. }
5. public class FunctionalInterfaceExample implements sayable{
6.     public void say(String msg){
7.         System.out.println(msg);
8.     }
9.     public static void main(String[] args) {
10.        FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
11.        fie.say("Hello there");
12.    }
13. }
```

Output:

Hello there

3. Java Default Methods

- Java provides a facility to create default methods inside the interface.
- Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

◇ Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

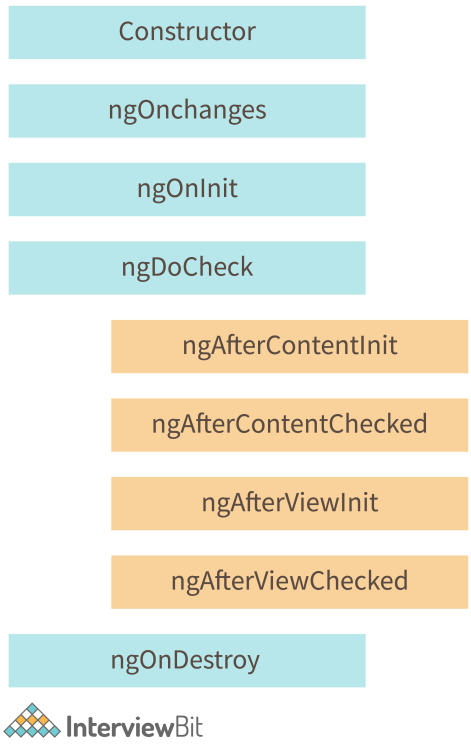
```
1. interface Sayable{
2.     // Default method
3.     default void say(){
4.         System.out.println("Hello, this is default method");
5.     }
6.     // Abstract method
7.     void sayMore(String msg);
8. }
9. public class DefaultMethods implements Sayable{
10.     public void sayMore(String msg){       // implementing abstract method
11.         System.out.println(msg);
12.     }
13.     public static void main(String[] args) {
14.         DefaultMethods dm = new DefaultMethods();
15.         dm.say();   // calling default method
16.         dm.sayMore("Work is worship");  // calling abstract method
17.
18.     }
19. }
```

Output:

```
Hello, this is default methodWork is worship
```

Angular Component Life Cycle Hooks

09 July 2024 12:06 PM



- Every component in Angular has a lifecycle, and different phases it goes through from the time of creation to the time it's destroyed.
- The NG on changes. Hook is called every time. Whenever there is a change detected in input properties of the component.

❖ So what is the purpose of NG on changes hook in Angular?

- So basically this hook is used to detect and respond to changes in input properties of an angular component and it is called whenever there are changes to input properties.

- When the NG on changes hook is defined, there is an argument passed in it which is of type simple changes interface object.
- It is used for viewing the changes and its related properties.
- When there is some change in the values as the simple changes.
- Is an object.
- It contains information in key and value pairs, where the key represents the properties that have changed and value represents the data that has changed.

1. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
2. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
3. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't

detect on its own.

4. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
5. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
6. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
7. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.
8. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

From <https://github.com/sudheerj/angular-interview-questions?tab=readme-ov-file#what-are-lifecycle-hooks-available>

- **ngOnChanges( )** This hook/method is called before ngOnInit and whenever one or more input properties of the component change. This method/hook receives a SimpleChanges object which contains the previous and current values of the property.
- **ngOnInit( )** This hook gets called once, after the ngOnChanges hook. It initializes the component and sets the input properties of the component.
- **ngDoCheck( )** It gets called after ngOnChanges and **ngOnInit** and is used to detect and act on changes that cannot be detected by Angular. We can implement our change detection algorithm in this hook. **ngAfterContentInit( )** It gets called after the first **ngDoCheck** hook. This hook responds after the content gets projected inside the component.
- **ngAfterContentChecked( )** It gets called after **ngAfterContentInit** and every subsequent **ngDoCheck**. It responds after the projected content is checked.
- **ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.
- **ngAfterViewChecked( )** It gets called after **ngAfterViewInit**, and it responds after the component's view, or the child component's view is checked.
- **ngOnDestroy( )** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.

From <https://www.interviewbit.com/angular-interview-questions/>

When the angular application starts, it first creates and renders the root component. Then, it creates and renders its Children's & their children. It forms a tree of components

Once Angular loads the components, it starts the process of rendering the view. To do that, it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc.

Angular lets us know when these events happen using lifecycle hooks. For Example:

ngOnInit when Angular initializes the component for the first time.

When a component's input property change, Angular invokes ngonChanges

If the component is destroyed, Angular invokes ngonDestroy

❖ What is change detection life cycle?

Change detection is the mechanism by which angular keeps the template in sync with the component.

1.Projected content: Projected content is that HTML content which replaces the <ng-content> directive in child component.

Child Component
<h2> Child Component</h2> <ng-content></ng-content>
Parent Component

<h1>Parent Component</h1>
<app-child>
This content is injected from parent
</app-child>

2 Input bound properties: These are those properties of a component class which is decorated with @Input() decorator.

Child Component

@Input() message: string

- Constructor of a Component

1.Life Cycle of a component begins, when Angular creates the component class. First method that gets invoked is class Constructor.

2.Constructor is neither a life cycle hook nor it is specific to Angular. It is a JavaScript feature. It is a method which gets invoked, when a class is created.

3.When a constructor is called, at that point, none of the components input properties are updated and available to use. Neither its child components are constructed. Projected contents are also not available.

4. Once Angular instantiates the class, it kick-start the first change detection cycle of the component.

1. ngOnChanges:

1.It is executed right at the start, when a new component is created, and it also gets executed whenever one of the bound input property changes.

2.Angular invokes ngOnChanges life cycle hook whenever any data-bound input property of the component or directive changes.

3.Input properties are those properties, which we define using the @Input decorator. It is one of the ways by which a parent component communicates with the child component.

e.g.
Child Component

@Input() message: string

Parent Component

<app-child [message]="message"> </app-child>

4.Initializing the Input properties is the first task that angular carries during the change detection cycle. And if it detects any change in input property, then it raises the ngOnChanges hook. It does so during every change detection cycle 4

5. This hook is not raised if change detection does not detect any changes.

2. ngOnInit:

   1.Angular raises the ngOnInit hook, after it creates the component and updates its input properties.

   2. This hook is fired only once and immediately after its creation (during the first change detection).

   3. This is a perfect place where you want to add any initialization logic for your component.

   4.Here you have access to every input property of the component. You can use them in http get requests to get the data from the back-end server or run some initialization logic etc.

   5.But remember that, by the time ngOnInit get's called, none of the child components or projected content are available at this point. Hence any properties we decorate with @ViewChild, @ViewChildren, @ContentChild & @ContentChildren will not be available to use

3. ngDoCheck:

   1. The Angular invokes the ngDoCheck hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties.

   2. For example, ngDoCheck will run if you clicked a button on the webpage which does not change anything. But still, it's an event.

   3. Angular invokes ngDoCheck it after the ngOnChanges & ngOnInit hooks.

   4.You can use this hook to Implement a custom change detection, whenever Angular fails to detect the changes made to Input properties.

   5. ngDoCheck is also a great method to use, when you want to execute some code on every change detection cycle.

4. ngAfterContentInit:

   1.ngAfterContentInit Life cycle hook is called after the Component's projected content has been fully initialized.

   2.Angular also updates the properties decorated with the @ContentChild and @ContentChildren before raising this hook. This hook is also raised, even if there is no content to project.

   3. The content here refers to the external content injected from the parent component via Content Projection.

   4. The Angular Components can include the ng-content element, which acts as a placeholder for

the content from the parent.

5. Parent injects the content between the opening & closing selector element. Angular passes this content to the child component

Child Component
```
<h2>Child Component</h2>
<ng-content></ng-content>
```

Parent Component
```
<h1>Parent Component</h1>
<app-child>
<p>This content is injected from parent</p>
</app-child>
```

5. ngAfterContentChecked

1.ngAfterContentChecked Life cycle hook is called during every change detection cycle after Angular finishes checking of component's projected content.

2.Angular also updates the properties decorated with the @ContentChild and @ContentChildren before raising this hook. Angular calls this hook even if there is no projected content in the component.

3. This hook is very similar to the ngAfterContentInit hook. Both are called after the external content is initialized, checked & updated.

4.Only difference is that ngAfterContentChecked is raised after every 4 change detection cycle. While ngAfterContentInit during the first change detection cycle.

- ngAfterContentInit and ngAfterContentChecked are component only hooks they are not applicable on directives.

6. ngAfterViewInit:

1.ngAfterViewInit hook is called after the Component's View & all its child views are fully initialized. Angular also updates the properties decorated with the @ViewChild & @ViewChildren properties before raising this hook.

2. The View here refers to the view template of the current component and all its child components & directives.

3.This hook is called during the first change detection cycle, where angular initializes the view for the first time

4. At this point all the lifecycle hook methods & change detection of all child components & directives are processed & Component is completely ready.

5. This is a component only hook.

7. ngAfterViewChecked:

1.The Angular fires this hook after it checks & updates the component's views and child views. This event is fired after the ngAfterViewInit and after that during every change detection cycle.

2.This hook is very similar to the ngAfterViewInit hook. Both are 2 called after all the child components & directives are initialized and updated.

3.Only difference is that ngAfterViewChecked is raised during every 3 change detection cycle. While ngAfterViewInit is raised during the first change detection cycle.

4.This is a component only hook.

8. ngOnDestroy:

1.If you destroy a component, for example, when you placed ngIf on a component, and this ng If then set to false, at that time, ngIf will remove that component from the DOM, at that time, ngonDestroy is called.

2.This method is the great place to do some cleanup work, because this 2 is called right before the objects will be destroyed itself by angular.

3. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

Mockito

09 November 2023 11:05 AM

Mocking:

Mocking is a process of developing the objects that act as the mock or clone of the real objects. In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes. Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.
The mocking technique is not only used in Java but also used in any object-oriented programming language. There are many frameworks available in Java for mocking, but Mockito is the most popular framework among them.
To mock objects, you need to understand the three key concepts of mocking, i.e., stub, fake, and mock. Some of the unit tests involve only stubs, whereas some involve fake and mocks.
The brief description of the mocking concepts is given below:
Stub: Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for state verification.
Fake: Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.
Mock: Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for behavior verification.

Mockito

Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications. Mockito was released as an open-source testing framework under the MIT (Massachusetts Institute of Technology) License. It internally uses the Java Reflection API to generate mock objects for a specific interface. Mock objects are referred to as the dummy or proxy objects used for actual implementations.
The main purpose of using the Mockito framework is to simplify the development of a test by mocking external dependencies and use them in the test code. As a result, it provides a simpler test code that is easier to read, understand, and modify. We can also use Mockito with other testing frameworks like JUnit and TestNG.
The Mockito framework was developed by upgrading the syntax and functionalities of
EasyMock framework. It was developed by a team of developers consisting of Szczepan Faber, Brice Dutheil, Rafael Winterhalter, Tim van der Lippe, and others. The stable or latest version of Mockito is version 3.0.6 was released in August 2019.

Benefits of Mockito:

No Handwriting
Safe Refactoring
Exception Support
Annotation Support
Order Support
Creation of Objects

What is Mock Object?

Mockito mock() method

In this post, We will learn How to set up Mockito Maven Dependency Workspace in Eclipse?

We can use org.mockito.Mockito class mock() method to create a mock object of a given class or interface. This is really the simplest way to mock an object.

We can mock an object using @Mock annotation also. It's really useful when we want to use the mocked object in multiple test methods because we want to avoid calling the mock() method multiple times.

When we use @Mock annotation then make sure that we call to initialize the mocked object. We can do this in the testing framework setup method or test method that is executed before the tests.

We have to add the following maven dependency in pom.xml to use Maven 3.0

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-core</artifactId>
<version>3.5.7</version>
<scope>test</scope>
</dependency>
```

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-core</artifactId>
<version>3.5.7</version>
<scope>test</scope>
</dependency>
```

Let's try to understand the above concept using a demo project

****

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.intotech</groupId>
	<artifactId>MockitoBasicExample</artifactId>
	<version>0.0.1-SNAPSHOT</version>

	<properties>
		<maven.compiler.target>8</maven.compiler.target>
		<maven.compiler.source>8</maven.compiler.source>
	</properties>

	<dependencies>
		<dependency>
		<groupId>mysql</groupId>
		<artifactId>mysql-connector-java</artifactId>
		<version>8.0.21</version>
		</dependency>

		<dependency>
		<groupId>org.mockito</groupId>
		<artifactId>mockito-core</artifactId>
		<version>3.5.7</version>
		<scope>test</scope>
		</dependency>

	</dependencies>
```

****

```
</project>
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.intotech</groupId>
	<artifactId>MockitoBasicExample</artifactId>
	<version>0.0.1-SNAPSHOT</version>

	<properties>
		<maven.compiler.target>8</maven.compiler.target>
		<maven.compiler.source>8</maven.compiler.source>
	</properties>

	<dependencies>
		<dependency>
		<groupId>mysql</groupId>
		<artifactId>mysql-connector-java</artifactId>
		<version>8.0.21</version>
		</dependency>

		<dependency>
		<groupId>org.mockito</groupId>
		<artifactId>mockito-core</artifactId>
		<version>3.5.7</version>
		<scope>test</scope>
		</dependency>

	</dependencies>

</project>
```

****

EmployeeDAO.java

```
package com.infotech.dao;

import java.util.List;
import com.infotech.model.Employee;
public interface EmployeeDAO {

	public abstract void createEmployee(Employee employee);
	public abstract Employee getEmployeeById(Integer employeeId);
	public abstract void updateEmployeeEmailById(String newEmail,Integer employeeId);
	public abstract void deleteEmployeeById(Integer employeeId);
	public abstract List<Employee> getAllEmployeesInfo();
}


package com.infotech.dao;

import java.util.List;
import com.infotech.model.Employee;
public interface EmployeeDAO {

	public abstract void createEmployee(Employee employee);
	public abstract Employee getEmployeeById(Integer employeeId);
	public abstract void updateEmployeeEmailById(String newEmail,Integer employeeId);
	public abstract void deleteEmployeeById(Integer employeeId);
	public abstract List<Employee> getAllEmployeesInfo();
```

```
}
```

****

EmployeeDAOImpl.java

```
package com.infotech.dao.impl;

import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

import com.infotech.dao.EmployeeDAO;
import com.infotech.model.Employee;
import com.infotech.util.DBUtil;

public class EmployeeDAOImpl implements EmployeeDAO {

	@Override
	public void createEmployee(Employee employee) {

		String SQL = "INSERT INTO employee_table(employee_name,email,salary,date_of_joining,bonus)VALUES(?,?,?,?,?)";
		try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

		ps.setString(1, employee.getEmployeeName());
		ps.setString(2, employee.getEmail());
		ps.setDouble(3, employee.getSalary());
		ps.setDate(4, new Date(employee.getDoj().getTime()));
		ps.setBigDecimal(5, employee.getBonus());

		int executeUpdate = ps.executeUpdate();

		if(executeUpdate ==1){
		System.out.println("Employee is crated..");
		}
		} catch (Exception e) {
		e.printStackTrace();
		}
	}

	@Override
	public Employee getEmployeeById(Integer employeeId) {
		Employee employee = null;
		String SQL = "SELECT *FROM employee_table WHERE employee_id=?";
		try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

		ps.setInt(1, employeeId);

		ResultSet rs = ps.executeQuery();
		if (rs.next()) {
		int empId = rs.getInt("employee_id");
		String eName = rs.getString("employee_name");
		String email = rs.getString("email");
		Double salary = rs.getDouble("salary");
		BigDecimal bonus = rs.getBigDecimal("bonus");
		Date date = rs.getDate("date_of_joining");

		employee = new Employee();
		employee.setEmployeeName(eName);
		employee.setBonus(bonus);
		employee.setDoj(date);
		employee.setEmail(email);
		employee.setEmployee_id(empId);
		employee.setSalary(salary);
		}

		} catch (Exception e) {
		e.printStackTrace();
		}

		return employee;
	}

	@Override
	public void updateEmployeeEmailById(String newEmail, Integer employeeId) {

	String SQL = "UPDATE employee_table set email=? WHERE employee_id=?;";
	try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

	ps.setString(1, newEmail);
	ps.setInt(2,employeeId);

	int executeUpdate = ps.executeUpdate();

	if(executeUpdate ==1){
	System.out.println("Employee email is updated..");
```

```
	}
	} catch (Exception e) {
	e.printStackTrace();
	}
	}

	@Override
	public void deleteEmployeeById(Integer employeeId) {

	String SQL = "DELETE FROM employee_table WHERE employee_id=?;";
	try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

	ps.setInt(1,employeeId);

	int executeUpdate = ps.executeUpdate();

	if(executeUpdate ==1){
	System.out.println("Employee is deleted with ID::"+employeeId);
	}
	} catch (Exception e) {
	e.printStackTrace();
	}

	}

	@Override
	public List<Employee> getAllEmployeesInfo() {

	List<Employee> empList = new ArrayList<>();
	String SQL = "SELECT *FROM employee_table";
	try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

	ResultSet rs = ps.executeQuery();
	while (rs.next()) {
	int empId = rs.getInt("employee_id");
	String eName = rs.getString("employee_name");
	String email = rs.getString("email");
	Double salary = rs.getDouble("salary");
	BigDecimal bonus = rs.getBigDecimal("bonus");
	Date date = rs.getDate("date_of_joining");

	Employee employee = new Employee();
	employee.setEmployeeName(eName);
	employee.setBonus(bonus);
	employee.setDoj(date);
	employee.setEmail(email);
	employee.setEmployee_id(empId);
	employee.setSalary(salary);

	empList.add(employee);
	}

	} catch (Exception e) {
	e.printStackTrace();
	}
	return empList;
	}
}
```

****

```
package com.infotech.dao.impl;

import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

import com.infotech.dao.EmployeeDAO;
import com.infotech.model.Employee;
import com.infotech.util.DBUtil;

public class EmployeeDAOImpl implements EmployeeDAO {

	@Override
	public void createEmployee(Employee employee) {

		String SQL = "INSERT INTO employee_table(employee_name,email,salary,date_of_joining,bonus)VALUES(?,?,?,?,?)";
		try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

		ps.setString(1, employee.getEmployeeName());
		ps.setString(2, employee.getEmail());
		ps.setDouble(3, employee.getSalary());
		ps.setDate(4, new Date(employee.getDoj().getTime()));
		ps.setBigDecimal(5, employee.getBonus());
```

```
        int executeUpdate = ps.executeUpdate();

        if(executeUpdate ==1){
        System.out.println("Employee is crated..");
        }
        } catch (Exception e) {
        e.printStackTrace();
        }
}

@Override
public Employee getEmployeeById(Integer employeeId) {
        Employee employee = null;
        String SQL = "SELECT *FROM employee_table WHERE employee_id=?";
        try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

        ps.setInt(1, employeeId);

        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
        int empId = rs.getInt("employee_id");
        String eName = rs.getString("employee_name");
        String email = rs.getString("email");
        Double salary = rs.getDouble("salary");
        BigDecimal bonus = rs.getBigDecimal("bonus");
        Date date = rs.getDate("date_of_joining");

        employee = new Employee();
        employee.setEmployeeName(eName);
        employee.setBonus(bonus);
        employee.setDoj(date);
        employee.setEmail(email);
        employee.setEmployee_id(empId);
        employee.setSalary(salary);
        }

        } catch (Exception e) {
        e.printStackTrace();
        }

        return employee;
}

@Override
public void updateEmployeeEmailById(String newEmail, Integer employeeId) {

String SQL = "UPDATE employee_table set email=? WHERE employee_id=?;";
try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

ps.setString(1, newEmail);
ps.setInt(2,employeeId);

int executeUpdate = ps.executeUpdate();

if(executeUpdate ==1){
System.out.println("Employee email is updated..");
}
} catch (Exception e) {
e.printStackTrace();
}
}

@Override
public void deleteEmployeeById(Integer employeeId) {

String SQL = "DELETE FROM employee_table WHERE employee_id=?;";
try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

ps.setInt(1,employeeId);

int executeUpdate = ps.executeUpdate();

if(executeUpdate ==1){
System.out.println("Employee is deleted with ID::"+employeeId);
}
} catch (Exception e) {
e.printStackTrace();
}

}

@Override
public List<Employee> getAllEmployeesInfo() {

List<Employee> empList = new ArrayList<>();
String SQL = "SELECT *FROM employee_table";
try(Connection connection = DBUtil.getConnection();PreparedStatement ps = connection.prepareStatement(SQL)) {

ResultSet rs = ps.executeQuery();
```

```
        while (rs.next()) {
        int empId = rs.getInt("employee_id");
        String eName = rs.getString("employee_name");
        String email = rs.getString("email");
        Double salary = rs.getDouble("salary");
        BigDecimal bonus = rs.getBigDecimal("bonus");
        Date date = rs.getDate("date_of_joining");

        Employee employee = new Employee();
        employee.setEmployeeName(eName);
        employee.setBonus(bonus);
        employee.setDoj(date);
        employee.setEmail(email);
        employee.setEmployee_id(empId);
        employee.setSalary(salary);

        empList.add(employee);
        }

        } catch (Exception e) {
        e.printStackTrace();
        }
        return empList;
        }
}
```

****

EmpoyeeService.java

```
package com.infotech.service;

import java.util.List;

import com.infotech.model.Employee;

public interface EmpoyeeService {

        public abstract void createEmployee(Employee employee);
        public abstract Employee fetchEmployeeById(Integer employeeId);
        public abstract void updateEmployeeEmailById(String newEmail,Integer employeeId);
        public abstract void deleteEmployeeById(Integer employeeId);
        public abstract List<Employee> fetchAllEmployeesInfo();
}
```

```
package com.infotech.service;

import java.util.List;

import com.infotech.model.Employee;

public interface EmpoyeeService {

        public abstract void createEmployee(Employee employee);
        public abstract Employee fetchEmployeeById(Integer employeeId);
        public abstract void updateEmployeeEmailById(String newEmail,Integer employeeId);
        public abstract void deleteEmployeeById(Integer employeeId);
        public abstract List<Employee> fetchAllEmployeesInfo();
}
```

EmployeeServiceImpl.java

```
package com.infotech.service;

import java.util.List;

import com.infotech.dao.EmployeeDAO;
import com.infotech.dao.impl.EmployeeDAOImpl;
import com.infotech.model.Employee;

public class EmployeeServiceImpl implements EmpoyeeService {

        private EmployeeDAO employeeDAO = new EmployeeDAOImpl();

        @Override
        public void createEmployee(Employee employee) {
                employeeDAO.createEmployee(employee);
        }

        @Override
        public Employee fetchEmployeeById(Integer employeeId) {
                return employeeDAO.getEmployeeById(employeeId);
        }

        @Override
        public void updateEmployeeEmailById(String newEmail, Integer employeeId) {
                employeeDAO.updateEmployeeEmailById(newEmail, employeeId);
        }
```

```
	@Override
	public void deleteEmployeeById(Integer employeeId) {
		employeeDAO.deleteEmployeeById(employeeId);
	}

	@Override
	public List<Employee> fetchAllEmployeesInfo() {
		return employeeDAO.getAllEmployeesInfo();
	}

}
```

****

```
package com.infotech.service;

import java.util.List;

import com.infotech.dao.EmployeeDAO;
import com.infotech.dao.impl.EmployeeDAOImpl;
import com.infotech.model.Employee;

public class EmployeeServiceImpl implements EmpoyeeService {

	private EmployeeDAO employeeDAO = new EmployeeDAOImpl();

	@Override
	public void createEmployee(Employee employee) {
		employeeDAO.createEmployee(employee);
	}

	@Override
	public Employee fetchEmployeeById(Integer employeeId) {
		return employeeDAO.getEmployeeById(employeeId);
	}

	@Override
	public void updateEmployeeEmailById(String newEmail, Integer employeeId) {
		employeeDAO.updateEmployeeEmailById(newEmail, employeeId);
	}

	@Override
	public void deleteEmployeeById(Integer employeeId) {
		employeeDAO.deleteEmployeeById(employeeId);
	}

	@Override
	public List<Employee> fetchAllEmployeesInfo() {
		return employeeDAO.getAllEmployeesInfo();
	}

}
```

****

DBUtil.java

```
package com.infotech.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {
	private static final String DB_DRIVER_CLASS = "com.mysql.cj.jdbc.Driver";
	private static final String DB_USERNAME = "root";
	private static final String DB_PASSWORD = "root";
	private static final String DB_URL = "jdbc:mysql://localhost:3306/test";

	private static Connection connection = null;
	static{
		try {
		Class.forName(DB_DRIVER_CLASS);
		connection = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
		} catch (ClassNotFoundException | SQLException e) {
		e.printStackTrace();
		}
	}

	public static Connection getConnection(){
		return connection;
	}
}
```

****

```
package com.infotech.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class DBUtil {
	private static final String DB_DRIVER_CLASS = "com.mysql.cj.jdbc.Driver";
	private static final String DB_USERNAME = "root";
	private static final String DB_PASSWORD = "root";
	private static final String DB_URL = "jdbc:mysql://localhost:3306/test";

	private static Connection connection = null;
	static{
		try {
		Class.forName(DB_DRIVER_CLASS);
		connection = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
		} catch (ClassNotFoundException | SQLException e) {
		e.printStackTrace();
		}
	}

	public static Connection getConnection(){
		return connection;
	}
}
```

****

Employee.java

```
package com.infotech.model;

import java.math.BigDecimal;
import java.util.Date;

public class Employee {

	private int employee_id;
	private String employeeName;
	private String email;
	private Double salary;
	private Date doj;
	private BigDecimal bonus;

	public int getEmployee_id() {
		return employee_id;
	}
	public void setEmployee_id(int employee_id) {
		this.employee_id = employee_id;
	}
	public String getEmployeeName() {
		return employeeName;
	}
	public void setEmployeeName(String employeeName) {
		this.employeeName = employeeName;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public Double getSalary() {
		return salary;
	}
	public void setSalary(Double salary) {
		this.salary = salary;
	}
	public Date getDoj() {
		return doj;
	}
	public void setDoj(Date doj) {
		this.doj = doj;
	}
	public BigDecimal getBonus() {
		return bonus;
	}
	public void setBonus(BigDecimal bonus) {
		this.bonus = bonus;
	}
	@Override
	public String toString() {
		return "Employee [employee_id=" + employee_id + ", employeeName=" + employeeName + ", email=" + email
		+ ", salary=" + salary + ", doj=" + doj + ", bonus=" + bonus + "]";
	}
}
```

****

```
package com.infotech.model;

import java.math.BigDecimal;
import java.util.Date;

public class Employee {
```

```
	private int employee_id;
	private String employeeName;
	private String email;
	private Double salary;
	private Date doj;
	private BigDecimal bonus;

	public int getEmployee_id() {
		return employee_id;
	}
	public void setEmployee_id(int employee_id) {
		this.employee_id = employee_id;
	}
	public String getEmployeeName() {
		return employeeName;
	}
	public void setEmployeeName(String employeeName) {
		this.employeeName = employeeName;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public Double getSalary() {
		return salary;
	}
	public void setSalary(Double salary) {
		this.salary = salary;
	}
	public Date getDoj() {
		return doj;
	}
	public void setDoj(Date doj) {
		this.doj = doj;
	}
	public BigDecimal getBonus() {
		return bonus;
	}
	public void setBonus(BigDecimal bonus) {
		this.bonus = bonus;
	}
	@Override
	public String toString() {
		return "Employee [employee_id=" + employee_id + ", employeeName=" + employeeName + ", email=" + email
		+ ", salary=" + salary + ", doj=" + doj + ", bonus=" + bonus + "]";
	}
}
```

****

ClientTest.java

```
package com.infotech.client;

import java.math.BigDecimal;
import java.util.Date;
import java.util.List;

import com.infotech.model.Employee;
import com.infotech.service.EmployeeServiceImpl;
import com.infotech.service.EmpoyeeService;

public class ClientTest {

	public static void main(String[] args) {
		EmpoyeeService empoyeeService = new EmployeeServiceImpl();

		Employee employee = getEmployee();
		empoyeeService.createEmployee(employee);
		//getEmployeeById(empoyeeService);
		//empoyeeService.updateEmployeeEmailById("sean.cs2020@gmail.com", 3);
		//empoyeeService.deleteEmployeeById(4);
		//getAllEmployeesInfo(empoyeeService);
	}

	private static void getAllEmployeesInfo(EmpoyeeService empoyeeService) {
		List<Employee> empList = empoyeeService.fetchAllEmployeesInfo();
		for (Employee employee : empList) {
		System.out.println(employee);
		}
	}

	private static void getEmployeeById(EmpoyeeService empoyeeService) {
		Employee employee2 = empoyeeService.fetchEmployeeById(1);
		if(employee2 != null){
		System.out.println(employee2);
```

```
		}else{
		System.out.println("Employee does not exist..");
		}
	}

	private static Employee getEmployee() {
		Employee employee = new Employee();
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

****

```
package com.infotech.client;

import java.math.BigDecimal;
import java.util.Date;
import java.util.List;

import com.infotech.model.Employee;
import com.infotech.service.EmployeeServiceImpl;
import com.infotech.service.EmpoyeeService;

public class ClientTest {

	public static void main(String[] args) {
		EmpoyeeService empoyeeService = new EmployeeServiceImpl();

		Employee employee = getEmployee();
		empoyeeService.createEmployee(employee);
		//getEmployeeById(empoyeeService);
		//empoyeeService.updateEmployeeEmailById("sean.cs2020@gmail.com", 3);
		//empoyeeService.deleteEmployeeById(4);
		//getAllEmployeesInfo(empoyeeService);
	}

	private static void getAllEmployeesInfo(EmpoyeeService empoyeeService) {
		List<Employee> empList = empoyeeService.fetchAllEmployeesInfo();
		for (Employee employee : empList) {
		System.out.println(employee);
		}
	}

	private static void getEmployeeById(EmpoyeeService empoyeeService) {
		Employee employee2 = empoyeeService.fetchEmployeeById(1);
		if(employee2 != null){
		System.out.println(employee2);
		}else{
		System.out.println("Employee does not exist..");
		}
	}

	private static Employee getEmployee() {
		Employee employee = new Employee();
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

****

DbScript

```
CREATE TABLE `employee_table` (
  `employee_id` int(11) NOT NULL auto_increment,
  `employee_name` varchar(60) NOT NULL,
  `email` varchar(45) NOT NULL,
  `salary` double default NULL,
  `date_of_joining` datetime default NULL,
  `bonus` decimal(10,0) default NULL,
  PRIMARY KEY  (`employee_id`)
) ;


INSERT INTO employee_table(employee_name,email,salary,date_of_joining,bonus)
		VALUES('Paul','paul.cs2009@yahoo.com',60000.00,'2017-05-17',400.00);



   CREATE TABLE `employee_table` (
```

```
 `employee_id` int(11) NOT NULL auto_increment,
 `employee_name` varchar(60) NOT NULL,
 `email` varchar(45) NOT NULL,
 `salary` double default NULL,
 `date_of_joining` datetime default NULL,
 `bonus` decimal(10,0) default NULL,
 PRIMARY KEY  (`employee_id`)
) ;


INSERT INTO employee_table(employee_name,email,salary,date_of_joining,bonus)
		VALUES('Paul','paul.cs2009@yahoo.com',60000.00,'2017-05-17',400.00);
```

****

InterfaceMethodMockingUsingmockMethodTest.java

```
package com.infotech.service;

import java.math.BigDecimal;
import java.util.Date;

import static org.mockito.Mockito.*;

import com.infotech.model.Employee;

public class InterfaceMethodMockingUsingmockMethodTest {

	private EmpoyeeService empoyeeService;

	public static void main(String[] args) {
		new InterfaceMethodMockingUsingmockMethodTest().testFetchEmployeeById();
	}


	public void testFetchEmployeeById() {
		empoyeeService = mock(EmpoyeeService.class);

		Employee employee = getEmployee();
		when(empoyeeService.fetchEmployeeById(10)).thenReturn(employee);

		System.out.println(empoyeeService.fetchEmployeeById(10));
	}


	private  Employee getEmployee() {
		Employee employee = new Employee();
		employee.setEmployee_id(10);
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

****

```
package com.infotech.service;

import java.math.BigDecimal;
import java.util.Date;

import static org.mockito.Mockito.*;

import com.infotech.model.Employee;

public class InterfaceMethodMockingUsingmockMethodTest {

	private EmpoyeeService empoyeeService;

	public static void main(String[] args) {
		new InterfaceMethodMockingUsingmockMethodTest().testFetchEmployeeById();
	}


	public void testFetchEmployeeById() {
		empoyeeService = mock(EmpoyeeService.class);

		Employee employee = getEmployee();
		when(empoyeeService.fetchEmployeeById(10)).thenReturn(employee);

		System.out.println(empoyeeService.fetchEmployeeById(10));
	}


	private  Employee getEmployee() {
		Employee employee = new Employee();
```

```
		employee.setEmployee_id(10);
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

****

InterfaceMethodMockingUsingMockAnnotationTest.java

```
package com.infotech.service;

import java.math.BigDecimal;
import java.util.Date;

import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import static org.mockito.Mockito.*;

import com.infotech.model.Employee;

public class InterfaceMethodMockingUsingMockAnnotationTest {

	@Mock
	private EmpoyeeService empoyeeService;

	public static void main(String[] args) {
		new InterfaceMethodMockingUsingMockAnnotationTest().testFetchEmployeeById();
	}

	public void testFetchEmployeeById() {
		//empoyeeService = mock(EmpoyeeService.class);
		MockitoAnnotations.openMocks(this);
		Employee employee = getEmployee();
		when(empoyeeService.fetchEmployeeById(10)).thenReturn(employee);

		System.out.println(empoyeeService.fetchEmployeeById(10));
	}


	private  Employee getEmployee() {
		Employee employee = new Employee();
		employee.setEmployee_id(10);
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

****

```
package com.infotech.service;

import java.math.BigDecimal;
import java.util.Date;

import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import static org.mockito.Mockito.*;

import com.infotech.model.Employee;

public class InterfaceMethodMockingUsingMockAnnotationTest {

	@Mock
	private EmpoyeeService empoyeeService;

	public static void main(String[] args) {
		new InterfaceMethodMockingUsingMockAnnotationTest().testFetchEmployeeById();
	}

	public void testFetchEmployeeById() {
		//empoyeeService = mock(EmpoyeeService.class);
		MockitoAnnotations.openMocks(this);
		Employee employee = getEmployee();
		when(empoyeeService.fetchEmployeeById(10)).thenReturn(employee);

		System.out.println(empoyeeService.fetchEmployeeById(10));
	}
```

```
	private  Employee getEmployee() {
		Employee employee = new Employee();
		employee.setEmployee_id(10);
		employee.setBonus(new BigDecimal(600));
		employee.setDoj(new Date());
		employee.setEmployeeName("KK");
		employee.setEmail("kk.cs2016@yahoo.com");
		employee.setSalary(50000.00);
		return employee;
	}
}
```

Now right click on either InterfaceMethodMockingUsingmockMethodTest.java or InterfaceMethodMockingUsingMockAnnotationTest.java class and select Run As then Java Application as shown in the below Image

Multithreading

30 October 2023 03:19 PM

Single Thread

```
package com.training.org;

class ThreadExample extends Thread{
	public void run() {
		System.out.println("Run method is called from ThreadExample class "+currentThread().getId());
	}
}


public class SingleThreadExample {
	public static void main(String[] args) {
	    ThreadExample th=new ThreadExample();
	    th.start();
	}
}
```

****

```
package com.training.org;

class ThreadExample extends Thread{
	private int sleepTime;
	// PrintThread constructor assigns name to thread
	// by calling superclass Thread constructor
	public ThreadExample(String name){
		super(name);
		// sleep between 0 and 5 seconds
		sleepTime = (int) ( Math.random() * 5000 );
		// display name and sleepTime
		System.err.println(
		"Name: " + getName() + ";  sleep: " + sleepTime );
	}
	// control thread's execution
	public void run()
	{
		// put thread to sleep for a random interval
		try {
		System.err.println( getName() + " going to sleep" );
		// put thread to sleep
		Thread.sleep( sleepTime );
		}
		catch(Exception e) {
		System.out.println(e.getMessage());
		}

	}

}

public class SingleThreadExample {
	public static void main(String[] args) {
		ThreadExample th1=new ThreadExample("Thread1");
		ThreadExample th2=new ThreadExample("Thread2");
		ThreadExample th3=new ThreadExample("Thread3");


	System.out.println("Starting Threads....");
	th1.start();
	th2.start();
	th3.start();

	}
}
```

****

```
package com.training.org;

class ThreadExample extends Thread{
        private int sleepTime;
        // PrintThread constructor assigns name to thread
        // by calling superclass Thread constructor
        public ThreadExample(String name){
                super(name);
                // sleep between 0 and 5 seconds
                sleepTime = (int) ( Math.random() * 5000 );
                // display name and sleepTime
                System.err.println(
                "Name: " + getName() + ";  sleep: " + sleepTime );
        }
        // control thread's execution
        public void run()
        {
                // put thread to sleep for a random interval
                try {
                System.err.println( getName() + " going to sleep" );
                // put thread to sleep
                Thread.sleep( sleepTime );
                }
      catch ( InterruptedException interruptedException ) {
         System.err.println( interruptedException.toString() );
        }

       // print thread name
       System.err.println( getName() + " done sleeping" );
        }

}

public class SingleThreadExample {
        public static void main(String[] args) {
                ThreadExample th1=new ThreadExample("Thread1");
                ThreadExample th2=new ThreadExample("Thread2");
                ThreadExample th3=new ThreadExample("Thread3");

//
        System.out.println("Starting Threads....");
        th1.start();
        th2.start();
        th3.start();

        }
}
```

****

THREAD PRIORITY

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority
(known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
3 constants defiend in Thread class:

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY
Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
```

```
 m1.start();
 m2.start();

 }
}
```

****

```
package com.training.org;

class ThreadExample extends Thread{
        private int sleepTime;
        // PrintThread constructor assigns name to thread
        // by calling superclass Thread constructor
        public ThreadExample(String name){
                super(name);
                // sleep between 0 and 5 seconds
                sleepTime = (int) ( Math.random() * 5000 );
                // display name and sleepTime
                System.err.println(
                "Name: " + getName() + ";  sleep: " + sleepTime );
        }
        // control thread's execution
        public void run()
        {
                System.err.println( getName() + " going to sleep" );
                // put thread to sleep
//                  Thread.sleep( sleepTime );

       // print thread name
       System.err.println( getName() + " done sleeping" );
        }

}

public class SingleThreadExample {
        public static void main(String[] args) {
                ThreadExample th1=new ThreadExample("Thread1");
                ThreadExample th2=new ThreadExample("Thread2");
                ThreadExample th3=new ThreadExample("Thread3");
                th1.setPriority(Thread.MIN_PRIORITY);
                th2.setPriority(Thread.MAX_PRIORITY);
                th3.setPriority(Thread.MAX_PRIORITY);
//
        System.out.println("Starting Threads....");
        th1.start();
        th2.start();
        th3.start();

        }
}
```

****

Q. Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
public class TestThreadTwice1 extends Thread{
 public void run(){
  System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
```

```
 t1.start();
 }
}
```

****

What if we call run() method directly instead start() method?

Each thread starts in a separate call stack.
Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
class TestCallRun1 extends Thread{
 public void run(){
  System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

****

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name
of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to return the name of a thread.
public void setName(String name): is used to change the name of a thread.
Example of naming a thread

```
class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  t1.start();
  t2.start();

  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

****

Current Thread

The currentThread() method returns a reference of currently executing thread.

public static Thread currentThread()
Example of currentThread() method

```
class TestMultiNaming2 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[]){
  TestMultiNaming2 t1=new TestMultiNaming2();
  TestMultiNaming2 t2=new TestMultiNaming2();

  t1.start();
  t2.start();
 }
}
```

****

JPA(Jakarta/Java Persistence API)

01 November 2023 10:11 AM

For theory refer Javatpoint

1.Create New Maven Project
2.Add following dependencies in pom.xml file

```
<dependencies>
        <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
        </dependency>
        <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.0.0</version>
        </dependency>

        <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.2.8.Final</version>
        </dependency>

        <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.11</version>
        </dependency>
    </dependencies>
```

3.create folder into src/main/resources
4.under that folder create xml file persistence.xml
5.add following code into that xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0">
        <persistence-unit name="StudentPU">
```

```
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <properties>
                <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/studentdata?useSSL=false" />
                <property name="hibernate.connection.driver_class"
                value="com.mysql.cj.jdbc.Driver" />
                <property name="hibernate.connection.username"
                value="shital" />
                <property name="hibernate.connection.password"
                value="shital@123" />
                <property name="hibernate.archive.autodetection"
                value="class" />
                <property name="hibernate.show_sql" value="true" />
                <property name="hibernate.format_sql" value="true" />
                <property name="hbm2ddl.auto" value="update" />
                </properties>
        </persistence-unit>
</persistence>
```

6.create student class in src/main/java

```
package JPAExample;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "STUDENT")

public class Student {
        @Id
        @Column(name = "id")
        private int id;

        @Column(name = "firstName")
        private String firstName;

        @Column(name = "lastName")
        private String lastName;

        @Column(name = "marks")
        private int marks;

        public Student() {
                System.out.println("Default constructor");
                id = 0;
                firstName = "";
                lastName = "";
                marks = 0;
        }

        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
```

```
	public int getMarks() {
		return marks;
	}

	public void setMarks(int marks) {
		this.marks = marks;
	}

	public Student(int id, String firstName, String lastName, int marks) {
		this.id = id;
		this.firstName = firstName;
		this.lastName = lastName;
		this.marks = marks;
	}

}
```

7.create table studentmanagement in same package

```
package JPAExample;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class StudentManagement {
	private static EntityManager em;

public static void main(String[] args) {
	EntityManagerFactory emf=Persistence.createEntityManagerFactory("StudentPU");
  em=emf.createEntityManager();

  //CREATE
//  em.getTransaction().begin();
//  Student student=new Student(101,"Eshan","Madake",80);
//  em.persist(student);
//  em.getTransaction().commit();


  //SEARCH
//  Student s=em.find(Student.class, 101);
//  if(s!=null) {
//  	System.out.println("Student first name="+s.getFirstName());
//  	System.out.println("Student ID="+s.getId());
//  	System.out.println("Student Last name="+s.getLastName());
//  	System.out.println("student marks="+s.getMarks());
//
//  }else {
//  	System.out.println("Record is not present in database.");
//  }


  //UPDATE

//  em.getTransaction().begin();
//  Student s=em.find(Student.class,101);
//  System.out.println("Student first name="+s.getFirstName());
//  	System.out.println("Student ID="+s.getId());
//  	System.out.println("Student Last name="+s.getLastName());
//  	System.out.println("student marks="+s.getMarks());
//
//  	s.setMarks(100);
//
//  	System.out.println("\n\nAfter Updation...");
//  	System.out.println("Student first name="+s.getFirstName());
//  	System.out.println("Student ID="+s.getId());
//  	System.out.println("Student Last name="+s.getLastName());
//  	System.out.println("student marks="+s.getMarks());
//
//  	em.getTransaction().commit();


  //DELETE
  em.getTransaction().begin();
  Student s=em.find(Student.class, 101);
```

```
em.remove(s);
em.getTransaction().commit();
System.out.println(em.getTransaction().isActive());

//close
em.close();
emf.close();
}
}
```

****

Practice example: Create table employee

```
CREATE TABLE EMPLOYEE(
EMPID INT NOT NULL,
NAME VARCHAR(20) DEFAULT NULL,
SALARY INT,
ADDRESS VARCHAR(200),
JOINING_DATE DATE
)
```

display years of experience of all the employees

find records whose salary is greater than the avg salary of all the employees

****

Another example

Create seperate project EmployeeManagement

Repeat steps 1to 6

```
CREATE TABLE employee_address(
e_pincode int,
e_city varchar(30),
e_state varchar(40),
employee_e_id int,
constraint pk_emp primary key (e_pincode),
constraint fk_deptno foreign key (employee_e_id) references employee(e_id)
);
```

```
CREATE TABLE Employee(
  e_id INT,
  e_name VARCHAR(10),
  constraint pk_emp primary key (e_id)
);
```

Persistence.xml file

```
<?xmlversion="1.0"encoding="UTF-8"?>
<persistencexmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
<persistence-unitname="StudentPU">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>com.training.com.Employee</class>
<class>com.training.com.Address</class>
```

```
    <class>com.training.com.Department</class>
    <properties>
    <propertyname="hibernate.connection.url"
    value="jdbc:mysql://localhost:3306/employeedata?useSSL=false"/>
    <propertyname="hibernate.connection.driver_class"
    value="com.mysql.cj.jdbc.Driver"/>
    <propertyname="hibernate.connection.username"
    value="root"/>
    <propertyname="hibernate.connection.password"
    value="root"/>
    <propertyname="hibernate.archive.autodetection"
    value="class"/>
    <propertyname="hibernate.show_sql"value="true"/>
    <propertyname="hibernate.format_sql"value="true"/>
    <propertyname="hbm2ddl.auto"value="update"/>
    </properties>
    </persistence-unit>
    </persistence>
```

```
package com.training.com;

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

import javax.persistence.ElementCollection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
public class Employee {
	@Id
	private int e_id;
	private String e_name;

	@ElementCollection
	private List<Address> address = new ArrayList<Address>();
	private Department dept;

	public Department getDepartment() {
		return dept;
	}

	public void setDepartment(Department dept) {
		this.dept = dept;
	}

//	private int dept_id;
//
//	public int getDepartment() {
//		return dept_id;
//	}
//	public void setDepartment(int dept_id) {
//		this.dept_id=dept_id;
//	}

	public int getE_id() {
		return e_id;
	}

	public void setE_id(int e_id) {
		this.e_id = e_id;
	}

	public String getE_name() {
		return e_name;
	}

	public void setE_name(String e_name) {
		this.e_name = e_name;
	}

	public List<Address> getAddress() {
		return address;
	}

	public void setAddress(List<Address> address) {
		this.address = address;
	}
```

```
}
```

```
package com.training.com;

import javax.persistence.Embeddable;

@Embeddable
public class Address {
private int e_pincode;
private String e_city;
private String e_state;

public int getE_pincode() {
return e_pincode;
        }

public void setE_pincode(int e_pincode) {
this.e_pincode = e_pincode;
        }

public String getE_city() {
return e_city;
        }

public void setE_city(String e_city) {
this.e_city = e_city;
        }

public String getE_state() {
return e_state;
        }

public void setE_state(String e_state) {
this.e_state = e_state;
        }

}
```

```
package com.training.com;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Snippet {
	private static EntityManager em;

	public static void main(String[] args) {

		EntityManagerFactory emf = Persistence.createEntityManagerFactory("StudentPU");
		em = emf.createEntityManager();

		em.getTransaction().begin();

		Address a1 = new Address();
		a1.setE_pincode(201301);
		a1.setE_city("Pune");
		a1.setE_state("Maharashtra");
//
//
//
//
//
		Address a2 = new Address();
		a2.setE_pincode(302001);
		a2.setE_city("Bangalore");
		a2.setE_state("Karanataka");
//


	Department d1=new Department();
		d1.setDept_id(101);
		d1.setDepartment_name("Science");
		d1.setDepartment_head("Yash");
//


		Employee e1 = new Employee();
```

```
                e1.setE_id(1);
                e1.setE_name("Vijay");
                e1.getAddress().add(a1);
        //


                Employee e2 = new Employee();
                e2.setE_id(2);
                e2.setE_name("John");
                e2.getAddress().add(a2);



                em.persist(e1);
                em.persist(e2);

                em.getTransaction().commit();

                em.close();
                emf.close();

            }

        }
```

Create table department
Deptid int
Deptname
Depthead int references from the employee eid

Create a single instance of department into employee and call it from snippet

To generate a report you need to simple run the Maven command:

mvn clean install test surefire-report:report

To run from eclipse you need to follow some steps.

Right click on project
Run As -> Run Configurations…
You will be prompted with the screen

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
        <version>2.19.1</version>
      </plugin>
    </plugins>
  </reporting>
```

Lambda Expression

31 October 2023 11:33 AM

```
package com.training.org;
import java.util.Scanner;
@FunctionalInterface
interface MyInterface{
double getPiValue();
}
public class LambdaAreaCalculation {
public static void main(String[] args) {

//with lambda expression
try (Scanner kb = new Scanner(System.in)) {
       out.println("Enter the value of radius");
int r = kb.nextInt();
            ref;        MyInterface
ref = () -> 3.1415;
       out.println("Area of Circle with given radius is = " + r * r * ref.getPiValue());
                }


//Without lambda expression...use implements MyInterface after class name
//        kb = new Scanner(System.in);
//int r = kb.nextInt();
//                MyInterface l1=new LambdaAreaCalculation();
//                System.out.println("Area of Circle with given radius is = " + r * r * l1.getPiValue());
        }

//        @Override
//        public double getPiValue() {
//                return 3.1415;
//        }
//

}
```

****

```
package com.training.org;

import java.util.Scanner;

@FunctionalInterface
interface MyInterface2 {
double getPiValue();
}

public class LambdaAreaCalculation2 {
public int show() {
return 100;
        }

public static void main(String[] args) {

// with lambda expression
try (Scanner kb = new Scanner(System.in)) {
       out.println("Enter the r value :");
int r = kb.nextInt();
             ref;       MyInterface2
ref = () -> 3.1415;
       out.println(ref);
       out.println(new LambdaAreaCalculation2().show());
       out.println("Area of Circle with given radius is = " + r * r *
ref.getPiValue());
  catch (Exception e) {
       out.println(e.getMessage());
                }

        }
}
```

****

```
package com.training.org;

import java.util.Scanner;
```

```
@FunctionalInterface
interface MyInterface3 {
double getPiValue(int a);
}

public class LambdaAreaCalculation3 {

public static void main(String[] args) {

// with lambda expression
try (Scanner kb = new Scanner(System.in)) {
       out.println("Enter the r value :");
int r = kb.nextInt();
                ref;       MyInterface3
ref = (int a) -> {
       out.println("This is lambda expression multiple statement");
return a * a * 3.1415;
                          };
       out.println(ref);System.

       out.println("Area of Circle with given radius is = " + ref.getPiValue(r));
  catch (Exception e) {
       out.println(e.getMessage());
                }

        }
}
```

****
**

```
package com.training.org;

import java.util.Scanner;

@FunctionalInterface
interface Sayable{
    String say(String message);
}

public class LambdaAreaCalculation4 {

public static void main(String[] args) {
        // You can pass multiple statements in lambda expression
     Sayable person = (message)-> {
         String str1 = "I would like to say, ";
         String str2 = str1 + message;
         return str2;
     };
         System.out.println(person.say("time is precious."));
 }
}
```

****

Functional Interface was introduced to Java while releasing its 8th version. The functional Interface is represented or identified, through its unique Functional Interface annotation. A functional interface is dedicated to storing only the default and static methods. These default and static methods will have implementations. Also, a functional interface must have one unimplemented abstract method in it.

****

```
package com.training.org;

@FunctionalInterface
interface cube {
int calculate(int a);
}

public class Functional {
public static void main(String args[]) {
int x = 5;
     c = (int a)cube a * a * a;
int result = c.calculate(x);
       out.println(result);
        }
}
```

****

```
package com.training.org;

@FunctionalInterface
interface Product {
float Mul(float x, float y);
}

public class LambdaMP {
public static void main(String[] args) {
        Mul1 = (float x, float y) -> (x * y);
       out.println(Mul1.Mul(2, 5));
        Mul2 = (float x, float y) -> (x * y);
       out.println(Mul2.Mul(100, 200));
        }
}
```

****

```
          package com.training.org;

          import java.util.*;

          public class LambdaExpressionExample {
          public static void main(String[] args) {

                      list = new ArrayList<String>();
          list.add("Eshan");
          list.add("Tanish");
          list.add("Anish");
          list.add("jai");

          list.forEach((n) -> System.out.println(n));
                  }
          }
```

****

```
   package com.training.org;

   import java.util.*;

   public class LambdaExpressionExample2 {
   public static void main(String[] args) {

               list = new ArrayList<String>();
   list.add("Eshan");
   list.add("Anand");
   list.add("Prashant");

   // first loop
   for (int i = 0; i < list.size(); i++) {
          out.println(list.get(i));
                   }

   // second loop
   for (String str : list) {
          out.println(str);System.
                   }

   // Third loop

   for (Iterator iterator = list.iterator(); iterator.hasNext();) {
          str = (String) iterator.next();
          out.println(str);System.
                   }

   // Fourth loop
   list.forEach((n) -> System.out.println(n));
           }
   }
```

****

This program is not running

```
package com.training1.org;

@FunctionalInterface
interface Addable {
int add(int a, int b);
}

public class LambdaExpressionExample3 {
public static void main(String[] args) {

// Lambda expression without return keyword.
a,b)ad1=(
                System.out.println(ad1.add(10,20))

// Lambda expression with return keyword.
              =(int)ad2{
return (a+b);
                };
                System.out.println(ad2.add(100,200));

                }
}
```

****

Default Methods for Interfaces
Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as virtual extension methods.

```
interface Formula {
  double calculate(int a);

  default double sqrt(int a) {
    return Math.sqrt(a);
  }
}
```

Besides the abstract method calculate the interface Formula also defines the default method sqrt. Concrete classes only have to implement the abstract method calculate. The default method sqrt can be used out of the box.

```
Formula formula = new Formula() {
  @Override
  public double calculate(int a) {
    return sqrt(a * 100);
  }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

The formula is implemented as an anonymous object. The code is quite verbose: 6 lines of code for such a simple calculation of sqrt(a * 100). As we'll see in the next section, there's a much nicer way of implementing single method objects in Java 8.

Lambda expressions
Let's start with a simple example of how to sort a list of strings in prior versions of Java:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
  @Override
  public int compare(String a, String b) {
    return b.compareTo(a);
  }
});
```

The static utility method Collections.sort accepts a list and a comparator in order to sort the elements of the given list. You often find yourself creating anonymous comparators and pass them to the sort method.
Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, lambda expressions:

```
Collections.sort(names, (String a, String b) -> {
  return b.compareTo(a);
});
```

As you can see the code is much shorter and easier to read. But it gets even shorter:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

For one line method bodies you can skip both the braces {} and the return keyword. But it gets even shorter:

```
names.sort((a, b) -> b.compareTo(a));
```

List now has a sort method. Also the java compiler is aware of the parameter types so you can skip them as well. Let's dive deeper into how lambda expressions can be used in the wild.
Functional Interfaces
How does lambda expressions fit into Java's type system? Each lambda corresponds to a given type, specified by an interface. A so called functional interface must contain exactly one abstract method declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.
We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method. To ensure that your interface meet the requirements, you should add the @FunctionalInterface annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract

method declaration to the interface.
Example:

```
@FunctionalInterface
interface Converter<F, T> {
   T convert(F from);
}
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Keep in mind that the code is also valid if the @FunctionalInterface annotation would be omitted.
Method and Constructor References
The above example code can be further simplified by utilizing static method references:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);   // 123
```

Java 8 enables you to pass references of methods or constructors via the :: keyword. The above example shows how to reference a static method. But we can also reference object methods:

```
class Something {
   String startsWith(String s) {
      return String.valueOf(s.charAt(0));
   }
}
Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

Let's see how the :: keyword works for constructors. First we define an example class with different constructors:

```
class Person {
   String firstName;
   String lastName;

   Person() {}

   Person(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

Next we specify a person factory interface to be used for creating new persons:

```
interface PersonFactory<P extends Person> {
   P create(String firstName, String lastName);
}
```

Instead of implementing the factory manually, we glue everything together via constructor references:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

We create a reference to the Person constructor via Person::new. The Java compiler automatically chooses the right constructor by matching the signature of PersonFactory.create.
Lambda Scopes
Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.
Accessing local variables
We can read final local variables from the outer scope of lambda expressions:

```
final int num = 1;
Converter<Integer, String> stringConverter =
      (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

But different to anonymous objects the variable num does not have to be declared final. This code is also valid:

```
int num = 1;
Converter<Integer, String> stringConverter =
      (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

However num must be implicitly final for the code to compile. The following code does not compile:

```
int num = 1;
Converter<Integer, String> stringConverter =
      (from) -> String.valueOf(from + num);
num = 3;
```

Writing to num from within the lambda expression is also prohibited.
Accessing fields and static variables
In contrast to local variables, we have both read and write access to instance fields and static variables from within lambda expressions. This behaviour is well known from anonymous objects.

```
class Lambda4 {
   static int outerStaticNum;
   int outerNum;

   void testScopes() {
      Converter<Integer, String> stringConverter1 = (from) -> {
         outerNum = 23;
         return String.valueOf(from);
      };

      Converter<Integer, String> stringConverter2 = (from) -> {
         outerStaticNum = 72;
         return String.valueOf(from);
      };
```

```
    }
}
```

Accessing Default Interface Methods

Remember the formula example from the first section? Interface Formula defines a default method sqrt which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods cannot be accessed from within lambda expressions. The following code does not compile:

```
Formula formula = (a) -> sqrt(a * 100);
```

Built-in Functional Interfaces

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like Comparator or Runnable. Those existing interfaces are extended to enable Lambda support via the @FunctionalInterface annotation.

But the Java 8 API is also full of new functional interfaces to make your life easier. Some of those new interfaces are well known from the Google Guava library. Even if you're familiar with this library you should keep a close eye on how those interfaces are extended by some useful method extensions.

Predicates

Predicates are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true
predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Functions

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"
```

Suppliers

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

Consumers

Consumers represent operations to be performed on a single input argument.

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparators

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```

Optionals

Optionals are not functional interfaces, but nifty utilities to prevent NullPointerException. It's an important concept for the next section, so let's have a quick look at how Optionals work.

Optional is a simple container for a value which may be null or non-null. Think of a method which may return a non-null result but sometimes return nothing. Instead of returning null you return an Optional in Java 8.

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();         // true
optional.get();               // "bam"
optional.orElse("fallback");  // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
```

Streams

A java.util.Stream represents a sequence of elements on which one or more operations can be performed. Stream operations are either intermediate or terminal. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequentially or parallely.

Streams are extremely powerful, so I wrote a separate Java 8 Streams Tutorial. You should also check out Sequency as a similiar library for the web.

Let's first look how sequential streams work. First we create a sample source in form of a list of strings:

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Collections in Java 8 are extended so you can simply create streams either by calling Collection.stream() or Collection.parallelStream(). The following sections explain the most common stream operations.

Filter

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (forEach) on the result. ForEach accepts a consumer to be executed for each element in the filtered stream. ForEach is a terminal operation. It's void, so we cannot call another stream operation.

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

```
stringCollection
   .stream()
   .sorted()
   .filter((s) -> s.startsWith("a"))
   .forEach(System.out::println);

// "aaa1", "aaa2"
```

Keep in mind that sorted does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of stringCollection is untouched:

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type. The generic type of the resulting stream depends on the generic type of the function you pass to map.

```
stringCollection
   .stream()
   .map(String::toUpperCase)
   .sorted((a, b) -> b.compareTo(a))
   .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.

```
boolean anyStartsWithA =
   stringCollection
      .stream()
      .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);     // true

boolean allStartsWithA =
   stringCollection
      .stream()
      .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);     // false

boolean noneStartsWithZ =
   stringCollection
      .stream()
      .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);     // true
```

Count

Count is a terminal operation returning the number of elements in the stream as a long.

```
long startsWithB =
   stringCollection
      .stream()
      .filter((s) -> s.startsWith("b"))
      .count();

System.out.println(startsWithB);    // 3
```

Reduce

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

```
Optional<String> reduced =
   stringCollection
      .stream()
      .sorted()
      .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Parallel Streams

As mentioned above streams can be either sequential or parallel. Operations on sequential streams are performed on a single thread while operations on parallel streams are performed concurrently on multiple threads.

The following example demonstrates how easy it is to increase the performance by using parallel streams.

First we create a large list of unique elements:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
   UUID uuid = UUID.randomUUID();
   values.add(uuid.toString());
}
```

Now we measure the time it takes to sort a stream of this collection.

Sequential Sort

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));
```

```
// sequential sort took: 899 ms
Parallel Sort
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms
```
As you can see both code snippets are almost identical but the parallel sort is roughly 50% faster. All you have to do is change stream() to parallelStream().
Maps
As already mentioned maps do not directly support streams. There's no stream() method available on the Map interface itself, however you can create specialized streams upon the keys, values or entries of a map via map.keySet().stream(), map.values().stream() and map.entrySet().stream().
Furthermore maps support various new and useful methods for doing common tasks.
```
Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {
   map.putIfAbsent(i, "val" + i);
}

map.forEach((id, val) -> System.out.println(val));
```
The above code should be self-explaining: putIfAbsent prevents us from writing additional if null checks; forEach accepts a consumer to perform operations for each value of the map.
This example shows how to compute code on the map by utilizing functions:
```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);           // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);    // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);   // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);           // val33
```
Next, we learn how to remove entries for a given key, only if it's currently mapped to a given value:
```
map.remove(3, "val3");
map.get(3);           // val33

map.remove(3, "val33");
map.get(3);           // null
```
Another helpful method:
```
map.getOrDefault(42, "not found");  // not found
```
Merging entries of a map is quite easy:
```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);           // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);           // val9concat
```
Merge either put the key/value into the map if no entry for the key exists, or the merging function will be called to change the existing value.
Date API
Java 8 contains a brand new date and time API under the package java.time. The new Date API is comparable with the Joda-Time library, however it's not the same. The following examples cover the most important parts of this new API.
Clock
Clock provides access to the current date and time. Clocks are aware of a timezone and may be used instead of System.currentTimeMillis() to retrieve the current time in milliseconds since Unix EPOCH. Such an instantaneous point on the time-line is also represented by the class Instant. Instants can be used to create legacy java.util.Date objects.
```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```
Timezones
Timezones are represented by a ZoneId. They can easily be accessed via static factory methods. Timezones define the offsets which are important to convert between instants and local dates and times.
```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```
LocalTime
LocalTime represents a time without a timezone, e.g. 10pm or 17:30:15. The following example creates two local times for the timezones defined above. Then we compare both times and calculate the difference in hours and minutes between both times.
```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false
```

```
long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);      // -3
System.out.println(minutesBetween);    // -239
```
LocalTime comes with various factory methods to simplify the creation of new instances, including parsing of time strings.
```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);      // 23:59:59

DateTimeFormatter germanFormatter =
   DateTimeFormatter
      .ofLocalizedTime(FormatStyle.SHORT)
      .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```
LocalDate
LocalDate represents a distinct date, e.g. 2014-03-11. It's immutable and works exactly analog to LocalTime. The sample demonstrates how to calculate new dates by adding or subtracting days, months or years. Keep in mind that each manipulation returns a new instance.
```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```
Parsing a LocalDate from a string is just as simple as parsing a LocalTime:
```
DateTimeFormatter germanFormatter =
   DateTimeFormatter
      .ofLocalizedDate(FormatStyle.MEDIUM)
      .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```
LocalDateTime
LocalDateTime represents a date-time. It combines date and time as seen in the above sections into one instance. LocalDateTime is immutable and works similar to LocalTime and LocalDate. We can utilize methods for retrieving certain fields from a date-time:
```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);     // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);         // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);   // 1439
```
With the additional information of a timezone it can be converted to an instant. Instants can easily be converted to legacy dates of type java.util.Date.
```
Instant instant = sylvester
      .atZone(ZoneId.systemDefault())
      .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate);    // Wed Dec 31 23:59:59 CET 2014
```
Formatting date-times works just like formatting dates or times. Instead of using pre-defined formats we can create formatters from custom patterns.
```
DateTimeFormatter formatter =
   DateTimeFormatter
      .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);
System.out.println(string);    // Nov 03, 2014 - 07:13
```
Unlike java.text.NumberFormat the new DateTimeFormatter is immutable and thread-safe.
For details on the pattern syntax read here.
Annotations
Annotations in Java 8 are repeatable. Let's dive directly into an example to figure that out.
First, we define a wrapper annotation which holds an array of the actual annotations:
```
@interface Hints {
   Hint[] value();
}

@Repeatable(Hints.class)
@interface Hint {
   String value();
}
```
Java 8 enables us to use multiple annotations of the same type by declaring the annotation @Repeatable.
Variant 1: Using the container annotation (old school)
```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```
Variant 2: Using repeatable annotations (new school)
```
@Hint("hint1")
@Hint("hint2")
class Person {}
```
Using variant 2 the java compiler implicitly sets up the @Hints annotation under the hood. That's important for reading annotation information via reflection.
```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);                   // null
```

```
Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);          // 2
```

Although we never declared the @Hints annotation on the Person class, it's still readable via getAnnotation(Hints.class). However, the more convenient method is getAnnotationsByType which grants direct access to all annotated @Hint annotations.

Furthermore the usage of annotations in Java 8 is expanded to two new targets:

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```