47.1.1. Select the correct answer 04:18

Functional Interfaces

An interface is said to be a functional interface when it contains only one abstract method, they can have only one functionality to exhibit. It's additionally recognized as **Single Abstract Method Interface,** which is also called **SAM interfaces**

The lambda expression is a new and important feature of Java that was included in Java SE 8. It is used to provide the implementation of an interface that has a functional interface.

In the case of the lambda expression, there is no need to define the method again for providing the implementation thus it saves a lot of code.

Select the correct option from the given option

- ( ) A functional interface can contain many abstract methods
- ( ) For lambda expression, we need to define the method again
- ( ) The functional interface also recognized as the Double Abstract Method Interface
- (●) A functional interface can only have one functionality to exhibit

47.1.2. Select the correct answer 00:11

Starting from Java SE 1.8, numerous interfaces have been transformed into functional interfaces and marked with the **@FunctionalInterface** annotation.

The **Runnable** interface has only one method, which is **run()**. **Comparable** has just the **compareTo()** method. **ActionListener** includes only the **actionPerformed()** method. **Callable** contains only the **call()** method.

Example of @FunctionalInterface

```
@FunctionalInterface
interface Calculator {
  int calculate(int a, int b);
}

public class CalculatorExample{
  public static void main(String[] args) {
    // Using a lambda expression to implement the calculate method
    Calculator addition = (a, b) -> a + b;

    // Using the functional interface
    int result = addition.calculate(5, 3);
    System.out.println("Addition Result: " + result);

    // Another implementation using a different lambda expression
    Calculator subtraction = (a, b) -> a - b;

    // Using the functional interface with a different operation
    result = subtraction.calculate(8, 2);
    System.out.println("Subtraction Result: " + result);
  }
}
```

Choose the correct option to implement multiplication lambda expression

- ( ) calculator multiplication = (a, b) -> a * b;
- (•) Calculator multiplication = (a, b) -> a * b;
- ( ) Calculator multiplication = (a, b) -> x * y;
- ( ) Calculator multiplication = (a) -> a * b;

Home Learn Anywhere [redacted] Support Logout

47.1.3. Write the code

03:19

lambda expressions to implement a user defined functional interface:

Implementation with documentation:

```
import java.util.*;
import java.lang.*;
import java.io.*;
// lambda expressions to implement a user defined functional interface.
//@FunctionalInterface you can use here just to tell compiler that only one abstract interface is d
// except this annotation also code works in similar way
interface Cube{
int Compute(int X);
}

class UserdefinedFunctionalInterface{
     public static void main(String args[]){
         Scanner S = new Scanner(System.in);
         System.out.print("Enter any number: ");
         int A = scan.nextInt();
// lambda expression definition for Compute method
         Cube C = (int X)->X*X*X;

 // Passing and returning parameters must be exactly
 // same as defined in the prototype
         int Result = C.Compute(A);
        System.out.println("Cube of value is “+Result);
    }
}
```

Sample Test Cases +

Explorer

Userdefin...

Submit

Plots

```
import java.util.*;

@FunctionalInterface
interface Cube {
    int compute(int x);
}

public class UserdefinedFunctionalInterface {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter any number: ");
        int inputNumber = scanner.nextInt();

        Cube cube = (int x) -> x * x * x;

        int result = cube.compute(inputNumber);
            System.out.println("Cube of value is " + result);
    }
}
```

< Prev Reset Submit Next >

47.1.4. Write the code

17:17

Functional Interfaces:

Key points:

1. Functional interface contains only one abstract method.
2. Functional interface can have only one functionality to exhibit.
3. Functional interface can have any number of default methods.

Note: lambda expressions can be used to represent the instance of a functional interface(lambda expressions are supported java 8 onwards).

Some of the examples of functional interfaces are:

Runnable
ActionListener
Comparable

Before Java 8 Java Code for creating functional interface implementation is given below:

```
import java.util.*;
import java.lang.*;
import java.io.*;

class FunctionalInterface {
    public static void main(String args[]) {
          new Thread(new Runnable(){  //Anonymous inner class object
             @Override
              public void run() {
                System.out.println("Creation of new thread");
             }
         }).start();
```

Sample Test Cases +

Explorer

Function...

Submit

```
// Type your code here as provided
public class FunctionalInterface {
    public static void main(String args[]) {
        new Thread(() -> System.out.println("Creation of new thread")).start();
    }
}
```

Plots

Prev Reset Submit Next

47.1.5. Write the code

02:28

Java 8 onwards demonstration to Implement functional interface using lambda expressions:
Implementation is:

```
import java.util.*;
import java.lang.*;
import java.io.*;

// Java 8 onwards demonstration to Implement functional interface using lambda expressions

class FunctionalInterfaceWithLambda {
    public static void main(String args[]) {
    // use of lambda expression to create object
        new Thread(()->{System.out.println("Creation of new thread");}).start();
    }
}
```

Resultant Output:
Creation of new thread

Note: @FunctionalInterface annotation is used to ensure that the functional interface can only have one abstract method, But if more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message.

Type the code given above or given in the editor and execute to check the functionality.

Sample Test Cases +

Explorer

Function...

Submit

Plots

```
//Type your code here as provided to be well versed
//with the concept
import java.util.*;

// Java 8 onwards demonstration to implement a functional interface using lambda expressions
class FunctionalInterfaceWithLambda {
    public static void main(String args[]) {
        // Use of lambda expression to create an object of the Runnable functional interface
        new Thread(() -> {
            System.out.println("Creation of new thread");
        }).start();
    }
}
```

Prev Reset Submit Next

48.1.1. Write the code

01:00

Few key points to lambda expressions:

1. Lambda expressions basically express instances of functional interfaces.
2. lambda expressions implement the only abstract function and therefore implement functional interfaces.
3. Lambda expressions can only be used to implement functional interfaces.

Note: lambda expressions are added in Java 8 and provide below functionalities.

1. Enable to treat functionality as a method argument that is code as data
2. A function that can be created without belonging to any class.
3. A lambda expression can be passed as an object and executed on demand.

Type below given code in the editor to run and understand the concept of the use of lambda expressions to implement a user defined functional interface.

```java
import java.util.*;
import java.lang.*;
import java.io.*;

//Use of lambda expressions to implement a user defined
//functional interface.

//Interface with single abstract method
interface FuncInterface{
    void AbstractFunction(int A);
    // Default non abstract function
    default void GeneralFunction(){
        System.out.println("Welcome");
```

Sample Test Cases +

Explorer

LambdaE...

Submit

```java
import java.util.*;

// User-defined functional interface with a single abstract method
interface FuncInterface {
    void AbstractFunction(int A);

    // Default non-abstract function
    default void GeneralFunction() {
        System.out.println("Welcome");
    }
}

public class LambdaExpression {
    public static void main(String args[]) {
        // Functional interface using lambda expression
        FuncInterface fobj = (int A) -> System.out.println(10 + A);

        // Calls to lambda expression
        fobj.AbstractFunction(15);
    }
}
```

Plots

48.1.2. Write the code

40:10

Complete the code in `LambdaExpression.java` file to print all odd elements in the array list using concept of `lambda expression`

Sample Test Cases +

Explorer

LambdaE...

Submit

```java
// }
// }
import java.util.ArrayList;

public class LambdaExpression {
    public static void main(String[] args) {
        // Sample array list
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(120);
        numbers.add(303);
        numbers.add(308);
        numbers.add(555);

        // Display all elements in the correct order
        for (Integer number : numbers) {
            System.out.println(number);
        }

        // Print all odd elements in the correct order using lambda expression
        System.out.println("All odd elements displayed");
        numbers.stream()
            .filter(number -> number % 2 != 0)
            .forEach(System.out::println);
    }
}
```

Plots

< Prev Reset Submit Next >

48.1.3. Select the correct answer

00:15

Predict the results given by below given code snippet.

```
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;

public class Main {
    public static void main(String[] args) {
        StringBuilder str1 = new StringBuilder("1578");
        System.out.println(str1.toStrings());
        StringBuilder str2 = str1.reverse();
        System.out.println(str2.toStrings());
        System.out.println(str1.toStrings());
    }
}
```

- 1578
  8751
  8751
- 1578
  8751
  1578
- 1578
  1578
  1578
- Error: cannot find symbol System.out.println(str2.toStrings())

49.1.1. Write the code

00:05

In a Java class, the fields are normally initialized in the constructors. However, there is another way of initializing them in an unnamed block also called **initializer block** or a **non-static initializer block**.

For example, in the below code you will notice that the field `value` is initialized to `3` in an initializer block.

Once the code is executed you will notice that the code snippets in the initializer blocks are executed (in the order in which they appear in the code) before the code in the constructor is executed. This is because the Java compiler copies all initializer blocks in the order they appear, into every constructor.

Since we are creating two instances of class `A` using the `new` keyword, the constructor is called twice and during the constructor call first, all the initializer blocks are executed.

If there is a block of code which has to be executed before the code inside any of the constructors is executed, such code can be placed in an initializer block.

Program:

```
package q11289;
public class NonStaticInitBlockDemo {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
    }
}
class A {
    private int value;
    //below is an example of non-static initialization block
    {
        value = 3;
        System.out.println("In non-static init block 1");
    }
    public A() {
```

- (●) In non-static init block 1
  In non-static init block 2
  In constructor
  value = 3
  In non-static init block 1
  In non-static init block 2
  In constructor
  value = 3
- ( ) In non-static init block 2
  In non-static init block 1
  In constructor
  value = 3
  In non-static init block 2
  In non-static init block 1
  In constructor
  value = 3
- ( ) In non-static init block 1
  In constructor
  value = 3
  In non-static init block 2
  In constructor
  value = 3
- ( ) In non-static init block 2
  In constructor
  value = 3
  In non-static init block 1
  In constructor
  value = 3

49.2.1. What are static init blocks?

00:11

Initializer blocks which are prefixed with the `static` keyword are called **static initializer block**.

Like the **non-static** initializer blocks, a class can have any number of static initializer blocks and they will be executed in the order in which they appear in the code.

The code inside the **non-static** initalizer block is copied inside the constructor and executed during every invocation of the constructor.

However, a **static** initializer block is executed only once when the class is loaded in memory. It has nothing to do with constructor invocation. Even if the constructor is not called to create an object instance, as long as the class is loaded the `static` initializer blocks get executed.

Since the classes are loaded only once by the system class loader, these `static` initializer blocks get executed only once.

Program :

```
public class StaticInitBlockDemo {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
    }
}
class A {
    //below is an example of static initialization block
    static {
        System.out.println("In static init block 1");
    }
    public A() {
        System.out.println("In constructor");
    }
    //below is an example of another static initialization block
    static {
```

- In static init block 2
  In static init block 1
  In constructor
- In static init block 1
  In static init block 2
  In constructor
  In constructor
- In static init block 1
  In static init block 2
  In constructor
- In static init block 2
  In static init block 1
  In constructor
  In constructor

50.1.1. Understanding static fields

05:17

In a Java class, the fields which are marked as `static` are called **static fields** and those that are **not marked as static** are called as **instance fields** or simply **fields**. For example, in the below code:

```
class A {
    static int a;
    int b;
}
```

`a` is a **static field**, while `b` is an **instance field**. Meaning, if we create two or three instances of class `A` as given below:

```
A a1 = new A();
A a2 = new A();
A a3 = new A();
```

All the three instances `a1`, `a2` and `a3` will have their individual copies of **instance field** `b`. However, all the three instances will share a single copy of the **static field** `a`.

Static fields can be initalized in static initializer blocks.

Instance fields are accessed using the dot `.` operator on the references of a class. However, the static fields are accessed using the dot `.` operator directly on the class name and not references (as shown in the below StaticFieldDemo class).

There is a bugged code provided for you in the editor. Your task is to debug the code in order to execute it successfully.Your output must match the displayed test case.

After execution, in the output you will notice that the value of `aStaticField` is the same in both `a1` and `a2`, while the values stored in `instanceField` differ.

Sample Test Cases +

Explorer

StaticFiel...

Submit

Plots

```
package q11291;
public class StaticFieldDemo {
    public static void main(String[] args) {
        A a1 = new A(3);
        A a2 = new A(4);
        // Setting the value of static field aStaticField
        A.aStaticField = 5;

        // Displaying the values
        System.out.println("a1 = " + a1.toString());
        System.out.println("a2 = " + a2.toString());
        System.out.println("A.aStaticField = " + A.aStaticField);
    }
}
class A {
    public static int aStaticField;
    private int instanceField;

    public A(int instanceField) {
        this.instanceField = instanceField;
    }
    @Override
    public String toString() {
        return "A [instanceField = " + instanceField + ", aStaticField = " + aStaticField + "]";
    }
}
```

50.2.1. Understanding static methods

03:27

In a Java class, the methods which are marked as `static` are called **static methods** and those that are **not marked as static** are called as **instance methods** or simply **methods**.

If you remember, the `main()` method is always marked as `static` so that the **JVM** can directly invoke it without creating an instance of the class which is being executed.

In other words, a method which is marked as `static` can be called or invoked directly without creating an instance of that **class**.

For example, in the below code:

```
class A {
    static int a;
    int b;
    public static int getA() {
        return a;
    }
    public int getB() {
        return b;
    }
}
```

`getA()` is a **static method**, while `getB()` is an **instance method**.

Static methods can access only static fields and other static members like methods and cannot access instance members. If a static method is trying to access an instance member (field or a method), the compiler will throw an error saying the instance field/method cannot be accessed from a static context.

Just like **static fields**, **static methods** too are accessed using the dot `.` operator on the class name

Sample Test Cases +

Explorer

StaticMet...

Submit

```
package q36048;
public class StaticMethodDemo {
    public static void main(String[] args) {
        A a1 = new A(3);
        A a2 = new A(4);
        System.out.println("a1 = " + a1);
        System.out.println("a2 = " + a2);
        System.out.println("A.getInstanceCount() = " + A.getInstanceCount());
    }
}
class A {
    private static int counter;
    private int instanceField;
    public A(int instanceField) {
        this.instanceField = instanceField;
        counter++;
    }
    public static int getInstanceCount() {
        return counter;
    }
    public String toString() {
        return "A [instanceField = " + instanceField + ", counter = " + counter + "]";
    }
}
```

Plots

50.3.1. Write the code

02:06

A Java class which is marked as static is called a static class.

A static class can be written only inside another class. For example:

```
class A {
    static class B {
        private int b;
        public B(int b) {
            this.b = b;
        }
    }
}
```

The `static` class `B` is called a **static nested class** since it is nested inside class `A`.

Just like static fields and static methods, **static nested classes** can be accessed using their other class name. For example instances of class `B` can be created as below:

```
A.B b = new B();
```

Algorithm

- Start the program.
- Create a class named StaticClassDemo.
- Define the main method within the StaticClassDemo class.
  - Create an instance of the nested static class A named a1 with a value of 3.
  - Create an instance of the nested static class A named a2 with a value of 4.
  - Print the string representation of a1 in the format "a1 = ".
  - Print the string representation of a2 "a1 = ".

Sample Test Cases +

Explorer

StaticCla...

Submit

Plots

```
package q36045;
public class StaticClassDemo {
    public static void main(String[] args) {
        // Create instances of the static nested class A
        A a1 = new A(3);
        A a2 = new A(4);
        // Print the string representation of a1 and a2
        System.out.println("a1 = " + a1);
        System.out.println("a2 = " + a2);
    }
    // Define the static nested class A
    static class A {
        private int value;
        // Constructor for class A
        public A(int value) {
            this.value = value;
        }
        // Override toString method to provide string representation
        @Override
        public String toString() {
            return "A [value = " + value + "]";
        }
    }
}
```

51.1.1. Understanding nested classes

04:58

In Java, we can write a class inside another class. Such classes are called nested classes.

We can classify the nested classes into **four** types :

1. **Inner classes**
2. **Static nested classes**
3. **Local classes**
4. **Anonymous classes**

We will learn about inner classes here and the rest in the later sections.

Nested classes which are not declared as static are called Inner classes.

For example:

```
class A { //this is called a top-level class
    class B { //this is an inner class
    }
}
```

Let us consider two top-level classes `A` and `B` such that class `B` needs access to private members of class `A` and class `B` is not used by any other class. In such a scenario it makes sense to make `B` as an inner class of `A` as given in the above example. By making it an inner class, class `B` will gain access to all the private members of class `A`, since `B` has also become an member of `A`.

Since **inner classes** are similar to instance fields, create an instance of an inner class we need to first have an instance of the outer class.

Observe the given code.

Sample Test Cases +

Explorer

Namer.java

Submit

Plots

```
package q11294;
public class Namer {
    private String name;
    public Namer(String name) {
        this.name = name;
    }
    class Prefixer {
        private String prefix;
        private Prefixer(String prefix) {
            this.prefix = prefix;
        }
        public String getCompleteName() {
            return prefix + " " + name;
        }
    }
    public static void main(String[] args) {
        Namer namer = new Namer("Doodle");
        Namer.Prefixer prefixer = namer.new Prefixer("Mr.");
        System.out.println(prefixer.getCompleteName());
    }
}
```

< Prev Reset Submit Next >

51.2.1. Understanding static nested classes

01:52

A nested class marked as static is called a **static nested class** and not an inner class.

Nested classes which are not declared as static are called inner classes.

For example:

```
class A { //this is called a top-level class
    class B { //this is an inner class
    }
    static class C { //this is a nested class
    }
}
```

In the above code,

1. `A` is called a **top-level class**
2. `B` is called an **inner class** and
3. `C` is called a **static nested class**

Note that the **static nested class** `C` can only access **static members** of class `A` and not `A's` instance members.

Unlike an **inner class** where we need an instance of **outer class** to create an instance of **inner class**, a **static nested class** can be directly created using the **new** keyword.

Algorithm:

- Define a class A.
- Define an inner class B inside A.
- In the constructor of class B, print the message "In inner class B's constructor".
- Define a static nested class C inside A.

Sample Test Cases +

Explorer

A.java

Submit

```
package q36047;
public class A {
    // Inner class B
    public class B {
        // Constructor of class B
        public B() {
            System.out.println("In inner class B's constructor");
        }
    }

    // Static nested class C
    public static class C {
        // Constructor of class C
        public C() {
            System.out.println("In static nested class C's constructor");
        }
    }

    public static void main(String[] args) {
        // Create an instance of inner class B
        A a = new A();
        A.B bInstance = a.new B();

        // Create an instance of static nested class C
        A.C cInstance = new A.C();
    }
}
```

Plots

52.1.1. Understanding local classes

01:50

A nested class which is declared inside a block (between an opening and closing brace) is called a local class.

Inner classes are also local classes. However, to differentiate between inner and local classes, we use the term local class to a class which is declared inside a method or a conditional or a loop block. For example:

```
class A { //this is called a top-level class
    class B { //this is an inner class
    }
    static class C { //this is an inner class
    }
    public void someMethodInClassA() {
        class D {
        }
    }
}
```

In the above code,

1. `A` is called a **top-level class**
2. `B` is called an **inner class**
3. `C` is called a **static nested class** and
4. `D` is called a **local class**

Local classes can access instance members of their enclosing class.

The visibility of the local class is restricted to the scope of the enclosing block (braces).

Note that local classes **cannot have static** methods or interfaces as members. We can have fields marked as static provided they are also marked as `final` (such fields are called as constants).

Sample Test Cases +

Explorer

A.java

Submit

```
package q11296;
public class A {
    private int value = 7;
    class B {
        private B() {
            System.out.println("In inner class B's constructor");
        }}
    static class C {
        private C() {
            System.out.println("In static nested class C's constructor");
        }
    }
    public void someMethodInClassA() {
        class D {
            private D() {
                System.out.println("In local class D's constructor");
                System.out.println("value = " + value);
            }
        }
        D d = new D();
    }
    public static void main(String[] args) {
        A a = new A();
        A.B b = new A().new B();
        A.C c = new A.C();
        a.someMethodInClassA();
    }
}
```

Plots

< Prev Reset Submit Next >

52.2.1. Understanding anonymous classes

00:20

An `anonymous class` is very similar to a `local class` except that an anonymous class combines **both the declaration and definition** of the class into a single expression statement.

Secondly an anonymous class usually implements an interface or extends a class.

For example the below code shows the **difference between** an **inner class** and an **anonymous class**:

```
interface Printer { //this is a top-level interface
    public void printMe();
}
class A {  //this is a normal top-level class
    public static void main(String[] args) {
        class PrinterImpl implements Printer { // an example of a normal local class
            public void printMe() {
                //do something....
            }
        }

        Printer myPrinter1 = new PrinterImpl();

        Printer myPrinter2 = new Printer() { // an example of an anonymous class
            public void printMe() {
                //do something....
            }
        };

        myPrinter1.printMe();
        myPrinter2.printMe();
    }
}
```

Sample Test Cases +

Explorer

A.java

Submit

```
package q11297;
interface Printer {
    public void printMe();
}
class A {
    public static void main(String[] args) {
        class PrinterImpl implements Printer {
            public void printMe() {
                System.out.println("I am in printMe method of the local class PrinterImpl instance");
            }
        }
        Printer myPrinter1 = new PrinterImpl();
        Printer myPrinter2 = new Printer() {
            public void printMe() {
                System.out.println("I am in printMe method of the anonymous class");
            }
        };
        myPrinter1.printMe();
        myPrinter2.printMe();
    }
}
```

Plots

52.2.2. Anonymous classes from interfaces and classes

00:23

Observe the partial code provided in the editor. It has two examples of anonymous classes. One is created by implementing an **interface** `Printer` and the other by extending a **class** named `Prefixer`.

Note that when an anonymous class is implementing an `interface`, the `new` keyword is prefixed to the `interface` name, which is followed by empty parenthesis.

And note that when an anonymous class is extending a **class**, the `new` keyword is prefixed to one of the existing constructors in that class.

Now, your task is to complete the Prefixer class. The functionality of the Prefixer class is described below :

- It has a protected String variable **prefix.**
- It has a constructor that takes a String parameter prefix and assigns it to the prefix variable.
- It has a method **getPrefixedName(String name)** that returns a string with the given name prefixed by the stored prefix.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Anonymo...

Submit

```java
package q11298;
interface Printer {
	public void printMe();
}
class Prefixer {
	protected String prefix;
		public Prefixer(String prefix) {
			this.prefix = prefix;
		}
		public String getPrefixedName(String name) {
			return prefix + " " + name;
		}
}
class AnonymousExample {
	public static void main(String[] args) {
		Printer printer = new Printer() {
			public void printMe() {
				System.out.println("printMe is called!");
			}
		};
		Prefixer doublePrefixer = new Prefixer("Hello") {
		public String getPrefixedName(String name) {
			return prefix + " " + prefix + " " + name;
		}
		};
		printer.printMe();
		System.out.println(doublePrefixer.getPrefixedName("James"));
	}
}
```

Plots

52.2.3. What an anonymous class can do and cannot do

00:08

An anonymous class **can**:

1. have its own `fields` and `custom methods` which may not be there in the interface it is implementing or the class it is extending
2. access `members` of its `enclosing class`
3. have `static fields` provided they are also marked as `final`
4. have `local classes`

An anonymous class **cannot**:

1. be reused. They are just like expressions, whose complete implementation is provided in the accompanying code block.
2. access `non-final local variables` that are declared in its enclosing block
3. have `static methods` or `interfaces`

Select all the correct statements from the below code:

```
interface Printer {
    public void printMe();
}
class Prefixer {
    protected String prefix;
    public Prefixer(String prefix) {
        this.prefix = prefix;
    }
    public String getPrefixedName(String name) {
        return prefix + " " + name;
    }
}
class AnonymousExample {
    public static void main(String[] args) {
        Printer printer = new Printer(String text) { // statement 1
            this.text = text; // statement 2
            public void printMe() {
                System.out.println("printMe is called!");
            }
            public static void printMe2() { // statement 3
                System.out.println("printMe2 is called!");
            }
```

- [x] Statements 1 and 2 will result in compilation errors.
- [ ] Statement 3 will not result in a compilation error, since anonymous classes can have custom methods.
- [ ] Statement 4 will result in a compilation error as it is taking an argument `"Hello"`, while there should have been only empty parenthesis.
- [ ] Statement 5 will result in compilation error, because it cannot declare extra methods other than those declared in the class `Prefixer`.

53.1.1. Understanding final variables/references

00:25

In Java, we can classify variables or references into 5 different types:

1. **Parameters** - these are the variables/references that appear in a method signature inside the parenthesis `()`.
2. **Local variables/references** - the variables/references that are declared inside a method are called local variables. Their scope/visibility is restricted only to their enclosing method block.
3. **Instance fields** - these are the fields declared in the class. They hold the state information of the class. Every instance of the class holds a copy of these fields.
4. **Static fields** - these are the fields declared in the class with a static keyword. Only one copy of such fields exists in memory and are shared by (or available to) all the instances of that class.
5. **Constants** - these are the variables/references declared in the class with both `static` and `final` keywords. Only one copy of such fields exist in memory like the **static fields**. However, these variables/references cannot be reassigned another value once they have been assigned a value.

Note:If a variable is only declared as `final` and not `static`, then it is not called a `constant`. It is simply called a **final variable**.

Select all the correct statements from the below code:

```
public class A {
    private int value1; // statement 1
    private final int value2; // statement 2
    private static int value3; // statement 3
    private static final int value4; // statement 4
}
```

- [ ] `value1` declared in **statement 1** is called a `parameter`.
- [ ] Only one copy of `value2` declared in **statement 2** is shared by all instances of class `A`.
- [ ] `value3` declared in **statement 3** is called a `instance field`.
- [x] `value4` declared in **statement 4** is called a `constant`.

53.1.2. Understanding final variables/references

00:32

Imagine a `variable` or a `reference` to be a **cup** (something like a coffee cup ☕).

We have **2 types** of **cups**. We will call the **cups** which are of type `primitive` as `variables` and the **cups** which are of type `classes` or `arrays` as `references`.

Meaning

```
int age;          // age        → primitive ☕
String firstName;   // firstName    → reference ☕
int[] marksArr;     // marksArr → reference ☕
String[] cityNamesArr;  // cityNamesArr → reference ☕
```

Once we mark one of the cups ☕ as `final`, they can be **assigned a value only once**. For example:

```
final int age;
age = 20; // this is a valid statement
age = 30; // compiler will give an error on this line stating the final variable age is already a
ssigned a value
```

The same applies to cups ☕ which are of type `references`.

In case of `references`, the cup ☕ once assigned a `reference` cannot be assigned a new `reference`. However, the object which the `reference` points to (or refers to) can undergo changes by operations (method calls) on the object.

Same is the case with `arrays`, since arrays are also objects. For example:

```
final int[] marksArr1 = {20, 30, 40};
final int[] marksArr2 = {10, 20, 30};
marksArr1[0] = 70;  // this is a valid statement,
            // since we are not changing the value in the cup ☕ marksArr1
            // we are changing the value inside the object pointed by the cup ☕ marksArr1
marksArr1 = marksArr2;   // compiler will throw an error on this line
```

- [ ] **Statements 1** and **2** result in compilation errors. Since both these fields are declared as `final`, they must initialized during declaration only.
- [ ] **Statement 3** will result in a compilation error. Since fields `id` and `name` are declared as `final` the third field `comments` must also be declared `final`.
- [x] **Statement 4** will **not** result in a compilation error, since field `comments` is not declared as `final`.
- [ ] **Statement 5** will result in a compilation error. Since reference `a2` is declared as `final` we cannot change the value of `comments` in `a2`.
- [x] **Statement 6** will result in a compilation error.

53.2.1. Understanding final methods

00:11

In **Java**, when we do not want a method in a class to be overridden in its subclasses, we declare that method as `final`.

We have many `final` methods in Object class. For example below code displays one of them:

```
public class Object {
    ...
    ...
    public final void wait() throws InterruptedException {
        wait(0);
    }
    ...
}
```

Select all the correct statements from the below code:

```
public class A {
    public void method1() { /*do something */ }     // statement 1
    public final void method2() { /*do something */ }   // statement 2
    public void method3() { /*do something */ }     // statement 3
}
public class B extends A {
    public void method1() { /*do something else */ }    // statement 4
    public void method2() { /*do something else */ }    // statement 5
    public final void method3() { /*do something else */ }  // statement 6
}
```

- [x] `method1` of class `A` in **statement 1** is overridden in class `B` in **statement 4**.
- [ ] **Statement 5** will not give any compilation error.
- [ ] Since class `A` has one method declared as `final`, we should also declare the class `A` as `final`.
- [ ] Statement 6 will result in a compilation error. Since the `method3` is not marked as `final` in class `A`, the subclass `B` cannot mark the method as `final` when it wants to override.

54.1.1. Understanding final class

00:09

In Java, when we do not want a class to ever have subclasses, we declare that class as `final`.

In Java we have many `final` classes. For example, like the String and all the wrapper classes like Integer, Float etc.

Select all the correct statements given below:

- [ ]
```
class A {} // statement 1
final class B extends A {} // statement 2
```

Statement 2 will result in a compilation error. Since class **A** is not marked as final, class B which extends class A, cannot mark itself as final.

- [ ] Object class declared in java.lang package, is a good example of final class.

Below is a valid code:

- [ ]
```
public final interface A {
}
```

- [x] We declare a class as final, when we do not want someone to change its implementation by subclassing it.

54.2.1. Understanding final keyword

00:09

Select all the correct statements from the below code:

```
public final class A {
    public void m1() {              // statement 1
        ...
    }
}
public class B extends A {              // statement 2
}
public class C {
    public void m2() {
        ...
    }
}
public class D extends C {
    public final void m2() {            // statement 3
        final int value = 2;
        final int[] valueArr = {2, 3, 4};
        final String text1 = "Cat on the wall";
        String text2 = text1;           // statement 4
        String text3 =  "Dog in the pool";
        text1 = text3;              // statement 5
        valueArr[1] = value;            // statement 6
    }
}
```

- [ ] **Statement 1** will result in a compilation error. Since class `A` is marked as `final` all its methods must also be marked as `final` so that its subclasses cannot modify the method implementation by overriding.
- [x] In **statement 2**, will result in compilation error. Since class `A` is declared as final it cannot be extended by class `B`.
- [ ] In **statement 3**, method `m2` of class `D` overrides the implementation provided in `m2` of class `C`. Since `m2` in class `C` is not declared `final`, `m2` of class `D` also should not be declared `final`.
- [ ] **Statement 4** will result in a compilation error. Since `text1` is declared as `final` it cannot be used in an any assignment statement.
- [x] **Statement 5** will result in a compilation error.
- [ ] **Statement 6** will result in a compilation error.

55.1.1. Understanding garbage collection

00:21

In memory management terminology, garbage means that portion of memory which was once occupied by objects and is currently no longer used by the program.

Garbage collection (**GC**) means reclaiming such memory so that, that space can be used for allocation to other objects.

In Java programming language, the JVM (Java Virtual Machine) which is responsible for running the Java application performs automatic garbage collections.

The automatic garbage collection in Java is performed by a special thread called Garbage Collector, which is a part of JVM.

Garbage Collector does not reclaim the memory of objects that are still in use. It only reclaims the memory of objects which are no longer in use.

An object is created and assigned memory when we use the `new` keyword followed by a constructor call.

The reference holds the address of the object in the memory.

That portion of memory in which the Java objects reside is called **heap**.

When the program no longer uses an object, such an object is called unreferenced object.

For example:

```
public class A {
    private String text1 = "Bali Islands";      // statement 1
    public String haveSomeFun() {
        String text2 = "Fly to " + text1;   // statement 2
        System.out.println(text2);      // statement 3
    }
}
```

- [ ] System.gc() method can be called to force garbage collection.

- [x]

```
Object a1 = new Object(); //statement 1
Object a2 = new Object(); //statement 2
a2 = a1;
```

In the above code, the initial object referred by `a2` in **statement 2** will be available for **GC** after the statement `a2 = a1;` is executed.

55.1.1. Understanding garbage collection

00:21

In memory management terminology, garbage means that portion of memory which was once occupied by objects and is currently no longer used by the program.

Garbage collection (**GC**) means reclaiming such memory so that, that space can be used for allocation to other objects.

In Java programming language, the JVM (Java Virtual Machine) which is responsible for running the Java application performs automatic garbage collections.

The automatic garbage collection in Java is performed by a special thread called Garbage Collector, which is a part of JVM.

Garbage Collector does not reclaim the memory of objects that are still in use. It only reclaims the memory of objects which are no longer in use.

An object is created and assigned memory when we use the `new` keyword followed by a constructor call.

The reference holds the address of the object in the memory.

That portion of memory in which the Java objects reside is called **heap**.

When the program no longer uses an object, such an object is called unreferenced object.

For example:

```
public class A {
    private String text1 = "Bali Islands";      // statement 1
    public String haveSomeFun() {
        String text2 = "Fly to " + text1;   // statement 2
        System.out.println(text2);      // statement 3
    }
}
```

- [ ] System.gc() method can be called to force garbage collection.

- [x]

```
Object a1 = new Object(); //statement 1
Object a2 = new Object(); //statement 2
a2 = a1;
```

In the above code, the initial object referred by `a2` in **statement 2** will be available for **GC** after the statement `a2 = a1;` is executed.

55.2.1. Understanding constructors and finalize method

00:07

In Java, the constructors are used to prepare a newly created object for use by initilaizing values passed to it as parameters.

Similarly, when the **GC** (Garbage Collector) decides to remove an object from memory, it calls the `finalize()` method on the object.

The `finalize()` method is declared in the Object class. Hence it is available in every class.

The default `finalize()` method available in the Object class does not do anything.

A Java class can override and provide its special implementation in the `finalize()` method.

Normally we do not override finalize to provide any special implementation. However, it is good to know that we do not call the `finalize()` method, it is the **GC** which calls it.

- [ ] Constructors are automatically called by the **GC** (Garbage Collector Thread) to create instances of new classes.
- [x] **GC** (Garbage Collector Thread) automatically calls the finalize method of the object whenever it is trying to reclaim the memory occupied by the object.

56.1.1. Understanding System class

01:22

Among the various classes available in `java.lang` package, System is one of the most commonly used classes after `String`.

The System class can neither be **instantiated** (since it has a private constructor) nor can be **extended** (since it is declared as final).

System class has three important `public` `static` fields:

1. **out** - is accessed as System.out. This `out` field is of type `PrintStream`. The `out` refers to the standard output stream.
2. **err** - is accessed as System.err. This `err` field is of type `PrintStream`. The `err` refers to the standard error stream.
3. **in** - is accessed as System.in. This `in` field is of type `InputStream`. The `in` refers to the standard input stream.

The System class provides a method called System.console(), which returns the java.io.Console object associated with the running Java process (Java Virtual Machine). Select all the correct statement given below.

- [ ] `java.lang` package must be imported in every class before we start using the `System` class.

- [ ] In the statement

```
System.out.println("Uranus");
```

`out` is a method present in `System` class.

- [x] In the statement

```
System.out.println("Uranus");
```

`println` is a method.

- [ ] In the statement

```
System.out.println("Uranus");
```

`println` is a method in `System` class.

56.1.2. Common methods in System class

01:09

Select all the correct statements for some of the common methods in `System` class. [Hint: Make sure to click on the method names and read the method documentation before you mark the answers].

- [x] currentTimeMillis() is a `static` method in `System` class. So we can directly call the method on the class name, as given below:

```
System.currentTimeMillis();
```

- [x] The currentTimeMillis() method in System class returns a `long` value representing the total time elapsed from the midnight of January 1, 1970 UTC until the current system time.

- [x] The currentTimeMillis() method in System class returns a `long` value with precision upto nanoseconds. We can use this method when we want to measure time difference between two events to the precision of nanoseconds.

- [x]

```
int[] sourceArr = {1, 3, 5, 7, 9};
int[] destinationArr = new int[5];
System.arraycopy(sourceArr, 1, destinationArr, 1, 2);
```

After the **arraycopy** method invocation, the values in **destinationArr** array will be `{0, 3, 5, 0, 0}`

56.2.1. Understanding println method

00:10

Select all the correct statements for the below code:

```
public class Demo {
    public static void main(String[] args) {
        System.out.print("Up ");
        System.out.print("up");
        System.out.print(" and away!");
    }
}
```

- [ ] There is no `print` method there is only `println`, so the above code will not compile.
- [ ] The `print` method used is in `System` class.
- [ ] The output will be as given below:

```
Up
up
and away!
```

- [x] The output will be as given below:

```
Up up and away!
```

56.3.1. Working with System time

01:01

In many situations, we would like to measure the time taken to perform a certain operation. The code in the given editor is an example which uses the methods present in the `System` class to measure the time taken.

Write a Java program to calculate the total sum of numbers from 0 to 999 and measure the time taken for the calculation using the **System.currentTimeMillis()** method.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

TimingEx...

Submit

```
// package q11312;
// public class TimingExample {
//	public static void main(String[] args) {
//		long startTime =
//
//		//complete the code..
//
//		long endTime =
//		System.out.println("Time taken in milliseconds = " + (endTime - startTime));
//	}
// }
package q11312;
public class TimingExample {
	public static void main(String[] args) {
		long startTime = System.currentTimeMillis();
		int total = 0;
		for (int i = 0; i < 10000; i++) {
		total = total + i;
		}
		long endTime = System.currentTimeMillis();
		System.out.println("Time taken in milliseconds = " + (endTime - startTime));
	}
}
```

Plots

< Prev Reset Submit Next >

57.1.1. Understanding the class named Arrays

03:34

Java collection framework has included a very useful utility class called Arrays.

Please note that the class we are talking about java.util.Arrays is completely different from the class java.lang.reflect.Array.

java.lang.reflect.Array class is part of `java.lang.reflect` package, and it allows for dynamic creation and manipulation of an array.

Arrays on other hand is a class in `java.util` package.

Arrays is a utility class. A Utility class is a class which contains static methods and has a private constructor so that it cannot be instantiated.

This class has many useful searching and sorting methods which operate on arrays.

Complete the code in the given editor to know the usage of a few commonly used methods like :

- sort(int[] array)
- equals(int[] array, int[] value)

Note in the below code the **main** method uses ellipses(...) instead of a **String[]**.

Sample Test Cases +

Explorer

ArraysDe...

Submit

Plots

```java
package q11313;
import java.util.*;
public class ArraysDemo {
    public static void main(String ... args) {

        Integer[] integerArr = {382, 34, 4, 223, 331};
        Arrays.sort(integerArr);
        System.out.println(Arrays.asList(integerArr));

        Integer[] integerArr2 = {4, 34, 223, 331, 382};
        System.out.println(Arrays.equals(integerArr, integerArr2));
        Integer[] integerArr3 = new Integer[10];
        Arrays.fill(integerArr3, 7);

    }

}
```

57.1.2. Understanding the class named Collections

00:57

Like the class Arrays which deals with arrays, Java collection framework also has a utility class called Collections which deals with collections.

Please note that the class java.util.Collections is completely different from the interface java.util.Collection.

java.util.Collection class is the root interface for all the collection classes.

Collections on other hand is a utility class, which contains static methods and has a private constructor so that it cannot be instantiated.

This class has many useful searching and sorting methods which operate on various collections.

Observe the given code in the editor to know the usage of a few commonly used methods like :

- shuffle(List<?> list)
- sort(List<T> list)
- fill(List<? super T> list, T obj)

Note in the below code the **main** method uses ellipses(...) instead of a **String[]**.

Your task is to complete the code by following the comment lines.

Sample Test Cases +

Explorer

Collectio...

Submit

```
//\t\t\t\t\t/*Use a sort method*/(integerList);
\t\t
//\t\t\tSystem.out.println("After sort integerList : " + integerList);
\t\t
//\t\t\t\t\t/*Use a fill method*/(integerList, 7);
\t\t
//\t\t\tSystem.out.println("After filling integerList with 7 : " + integerList);
//\t}
//}
package q11314;
import java.util.*;
public class CollectionsDemo {
\t\tpublic static void main(String ... args) {
\t\t\t\t\t\t\t\t\t\t\tList<Integer> integerList = Arrays.asList(3, 33, 333, 3333, 33333, 333333);
\t\t\t\t\t\t\t\t\t\t\tSystem.out.println("integerList : " + integerList);
\t\t\t\t\t\t\t\t\t\t\tCollections.shuffle(integerList, new Random(1));
\t\t\t\t\t\t\t\t\t\t\tSystem.out.println("After shuffle integerList : " + integerList);
\t\t\t\t\t\t\t\t\t\t\tCollections.sort(integerList);
\t\t\t\t\t\t\t\t\t\t\tSystem.out.println("After sort integerList : " + integerList);
\t\t\t\t\t\t\t\t\t\t\tCollections.fill(integerList, 7);
\t\t\t\t\t\t\t\t\t\t\tSystem.out.println("After filling integerList with 7 : " + integerList);
\t\t}
}
```

Plots

< Prev Reset Submit Next >

57.2.1. Understanding Date and Time classes

00:26

The central classes in Java for working with Date and Time are Date, DateFormat and Calendar.

Date and Calendar classes are present in `java.util` package, while DateFormat is present in `java.text` package.

Starting with **Java version 8**, many new classes and enhancements were introduced for handling date and time, these are bundled in a new package named `java.time`.

We will learn about the new classes introduced in **Java version 8** later.

Since a lot of code written before `Java 8`, heavily use Date, DateFormat and Calendar classes, we will first learn about them.

1. Date - represents an instance in time (stored as a primitive `long`). For example this moment. Though the name might suggest that it represents only a calendar date, like 7th July 1977, we should also remember that it actually has the time component to the precision of a millisecond.
2. DateFormat - provides the formating for dates and times on an given Locale
3. Calendar - provides methods to work with the instance of time represented by Date.

It is important to note that earlier Unix systems used a 32-bit signed integer to store time. They started counting time with the value 0 representing 1970-1-1 00:00:00. This is referred as the epoch time.

The time component represented by the Date object counts the milliseconds passed from the above mentioned epoch time.

```
package q11315;
import java.util.*;
public class DateDemo {
    public static void main(String ... args) {
        Date thisMoment = new Date();
        long millisecondsSinceEpochStart = thisMoment.getTime();
        System.out.println("This Moment : " + thisMoment);
        System.out.println("Total milli seconds from epoch to this moment : " + millisecondsSinceEp
    }
```

- (●) This Moment : [Current Date and Time]
  Total milli seconds from epoch to this moment : [Milliseconds since epoch]
- ( ) This Moment : [Current Date and Time]
  Total Nano seconds from epoch to this moment : [Nanoseconds since epoch]
- ( ) This Moment : [Current Date and Time]
  Total micro seconds from epoch to this moment : [Microseconds since epoch]
- ( ) This Moment : [Current Date and Time]
  Total seconds from epoch to this moment : [seconds since epoch]

57.2.2. Understanding Date class

00:23

The Date class is a simple wrapper over an instance of time, represented as a `long`, just like the **Integer** class is a wrapper over an **int** value.

A Date object can be created using one of the below two constructors:

1. Date() - creates a new instance of Date object which represents the time at which it is created.
2. Date(long time) - creates a new instance of Date object which represents the specified number of milliseconds specified by the variable `time`, starting from the "the epoch", namely January 1, 1970, 00:00:00 GMT

The various methods in the Date class like getDay(), getHour(), getYear() etc., are all `deprecated` **and should not be used**.

We should instead use their alternatives provided in the Calendar class. We will learn more about the Calendar class later.

Date class has one useful method called getTime() which returns the number of milliseconds starting from January 1, 1970, 00:00:00 GMT represented by a date object as a `long`.

Note: `System` class also has a method called System.currentTimeMillis() which will return the current time as milliseconds from the **epoch time** (January 1, 1970, 00:00:00 GMT) as a `long` value.

Complete the code in the given editor by following the comment lines.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

DateExa...

Submit

Plots

```
//      Thread.sleep(1000);

//      //Use a method to return the current time as milliseconds from the epoch time

//      if (newCurrentTimeInMillis - oldTimeInMillis >= 1000) {
//          System.out.println("This thread resumed after 1 or more seconds");
//      }
//  }
//  }
package q11397;
import java.util.*;
public class DateExample {
    public static void main(String ... args) throws InterruptedException {
        System.out.println("Sleeping for 1 second...");
        Date timeAsDate = new Date();
        long oldTimeInMillis = timeAsDate.getTime();
        Thread.sleep(1000);
        long newCurrentTimeInMillis = System.currentTimeMillis();
        if (newCurrentTimeInMillis - oldTimeInMillis >= 1000) {
            System.out.println("This thread resumed after 1 or more seconds");
        }
    }
}
```

< Prev Reset Submit Next >

57.2.3. Understanding DateFormat class

00:33

DateFormat is an abstract class which provides methods for parsing `text to Date` objects and formating `Date objects to text`.

DateFormat class has static methods to create instance of `DateFormat` objects.

```
DateFormat df1 = DateFormat.getInstance();
// the above code creates a date and time formatter instance which uses the SHORT style

DateFormat df2 = DateFormat.getDateInstance();
// the above line creates a date foramatter instance with system's Locale

DateFormat df3 = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);
// the above line creates a date formatter instance with French Locale in long format
```

A Locale object represents a specific geographical, political, or cultural region and their attributes. For example, the month August is called **August** in English and **Août** in French.

The Locale class allows for displaying numbers, currencies, date and time information according to user's native language, country and cultural settings.

SimpleDateFormat is one of the most useful implementation sub-class of DateFormat class.

Note that the formatting classes like `DateFormat` and `SimpleDateFormat` are present in the `java.text` package.

Write a class named `DateFormatInJapan` with a static method `getJapaneseShortDate()`. The method should accept one parameter of type `Date`. It should return a formatted date as **String** from the passed parameter using **JAPAN** locale in **SHORT** format.

Sample Test Cases +

Explorer

DateForm... | DateForm... | Submit | Plots

```
package q11398;
import java.util.*;
import java.text.*;
public class DateFormatInJapan {
	public static String getJapaneseShortDate(Date date) {
		Format f=new SimpleDateFormat("yyyy/MM/dd");
		String ans=f.format(date);
		return ans;

	}
}
```

< Prev | Reset | Submit | Next >

57.2.4. Understanding SimpleDateFormat class

00:15

SimpleDateFormat is a concrete implementation of the abstract class DateFormat.

An instance of SimpleDateFormat class can be created using one of the below four constructors:

1. SimpleDateFormat() - creates default SimpleDateFormat object with default locale.
2. SimpleDateFormat(String pattern) - creates a SimpleDateFormat object with the given pattern.
3. SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols) - creates a SimpleDateFormat object with the given pattern and custom date format symbols.
4. SimpleDateFormat(String pattern, Locale locale) - creates a SimpleDateFormat object with the given pattern and locale.

The pattern string is a sequence of characters which have special meanings. Click here to view all the pattern characters with their meanings under the section titled ***Date and Time Patterns***.

After a SimpleDateFormat object is created applyPattern(String pattern) method can be used to modify the pattern.

Below are the two important methods of DateFormat that can be called on a SimpleDateFormat instance:

1. format(Date date) - formats the Date object as text.
2. parse(String dateText) - creates a Date object by parsing the dateText.

Note that if text has to be included as it is in the pattern, the text has to be surrounded in **single quotes**.

Write a class with name `SimpleDateFormateDemo`. In the main method create a Date object by parsing the given dateText `15-08-1947` and print the same.

Sample Test Cases +

Explorer

SimpleDa...

Submit

```java
package q11399;
import java.util.*;
import java.text.*;
public class SimpleDateFormatDemo {
	public static void main(String ... args) throws ParseException {
		String dateText = " ";
		SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy");
		// parse the dateText here and print.
		Date Nashe = simpleDateFormat.parse("15-08-1947");
		System.out.println("parsedDate : "+ Nashe);
	}
}
```

Plots

57.2.5. Understanding Calendar class

00:25

Calendar is an `abstract` class, which means we cannot create an instance of a Calendar class directly.

The Calendar class provides a static method called Calendar.getInstance(), which returns an object of type Calendar.

```
Calendar currentTime = Calendar.getInstance();
```

The above `getInstance()` method creates a Calendar object which is initialized with the current system time at that moment.

In most cases `getInstance()` returns an instance of GregorianCalendar which is a subclass of `Calendar`.

Also note that the above method uses the system's default locale, we can use another method Calendar.getInstance(Locale locale) to create a Calendar object with a specific locale. (We will learn about Locale when we learn about DateFormat)

Calendar also has two methods to reset the time component it represents, after a calendar instance is created.

1. setTime(Date newTime) - (note it has a corresponding getTime() method which returns a Date object for the Calendar's time component.)
2. setTimeInMillis(long timeInMilliSeconds)

Calendar has a generic method called get(int field). The field value can be any of the fields declared in the Calendar class like: Calendar.YEAR, Calendar.MONTH, Calendar.DAY_OF_MONTH, Calendar.WEEK_OF_MONTH, Calendar.DAY_OF_YEAR, Calendar.HOUR, Calendar.HOUR_OF_DAY, Calendar.MINUTE, Calendar.SECOND, Calendar.MILLISECOND, etc.

Sample Test Cases +

Explorer

Calendar... Calendar...

Submit

Plots

```java
package q11589;
import java.util.*;
import java.text.SimpleDateFormat;
import java.text.ParseException;
public class CalendarDemo {
    public String setTime(String dateString) throws ParseException {
        // use the correct format string as argument for the constructor
        SimpleDateFormat sd = new SimpleDateFormat("MMM-dd-yyyy");

        // parses text from the beginning of the given string to produce a date
        Date date =  sd.parse(dateString);


        Calendar currentTime = Calendar.getInstance();

        //// configuring the current object to the value of date
        currentTime.setTime(date);

        // format method on SimpleDateFormat returns a string representation of the passed date
        return sd.format(currentTime.getTime());
    }
}
```

< Prev Reset Submit Next >

57.2.6. Understanding Calendar class

00:15

Calendar class provides many useful methods for date and time manipulations.

1. roll(int field, int amount)
2. add(int field, int amount)
3. compareTo(Calendar anotherCalendar) - returns 1 if greater, 0 if equal and -1 if lesser

Write the missing code in the below program.

In this program `CalendarAddition` class is created with a method name `addDays` which passes two arguments to add the specified amount of time to the given calendar field. Parse the string using simpleDateFormate. In `add` method add the number of days to the calendar field.

Consider the following example for your understanding.

```
Adding (10) days to Feb-19-2005 gives Mar-01-2005
```

Note: Please don't change the package name.

Sample Test Cases +

Calendar... Calendar... Submit

```java
package q11590;
import java.util.*;
import java.text.SimpleDateFormat;
import java.text.ParseException;
public class CalendarAddition {
    public String addDays(String dateString, int days) throws ParseException {
        // use the correct format string as argument for the constructor
        SimpleDateFormat sd = new SimpleDateFormat("MMM-dd-yyyy");

        // parses text from the beginning of the given string to produce a date
        Date date = sd.parse(dateString);

        Calendar cal = Calendar.getInstance();

        // configuring the cal object to the value of date
        cal.setTime(date);

        // supply the correct arguments to the add method below
        cal.add(Calendar.DATE,days);

        // format method on SimpleDateFormat returns a string representation of the passed date
        return sd.format(cal.getTime());
    }
}
```

Explorer Plots

< Prev Reset Submit Next >

57.2.7. Choose the correct code

00:09

Given :

```
import java.util.Date;
import java.text.DateFormat;

DateFormat df;
Date date = new Date();
// insert code here
String s = df.format(date);
```

Which code fragment, inserted at `// insert code here`, allows the code to compile?

[Hint: Click on Date and DateFormat to explore the methods.]

- [ ] df = new DateFormat();
- [ ] df = Date.getFormat();
- [ ] df = date.getFormat();
- [ ] df = DateFormat.getFormat();
- [x] df = DateFormat.getInstance();

57.2.8. Understanding classes in java.time package

00:17

The java.time package introduced in **Java 8** contains many new classes apart from replacements for the existing `Date`, `DateFormat` and `Calendar` classes.

For a complete list of new classes in **java.time** package click here.

Below are list of a few important classes:

1. Instant - represents a moment on the time-scale. Unlike the `java.util.Date` which measures till milliseconds, an Instant records even nanoseconds.
2. LocalDate - represents a date without the time component
3. LocalTime - represents a time without the date component
4. LocalDateTime - represents an object with both date and time components
5. ZonedDateTime - represents a complete date-time object with time-zone

Complete the code in the given editor by following the comment lines.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

LocalDat...

Submit

Plots

```
// import java.time.temporal.*;
// public class LocalDateDemo {
//	public static void main(String ... args) {

//		//Get a local date of May 23,1995

//		TemporalAdjuster adjuster = TemporalAdjusters.next(DayOfWeek.SUNDAY);
//		LocalDate nextSunday = javaBirthday.with(adjuster);

//		/*print the statement as follows:
//		Java's birthday was on DATE, and the cake was cut on next Sunday : DATE*/
//	}
//}
package q11592;
import java.time.*;
import java.time.temporal.*;
public class LocalDateDemo {
	public static void main(String ... args) {
		LocalDate javaBirthday = LocalDate.of(1995, Month.MAY, 23);
		TemporalAdjuster adjuster = TemporalAdjusters.next(DayOfWeek.SUNDAY);
		LocalDate nextSunday = javaBirthday.with(adjuster);
		System.out.printf("Java's birthday was on %s, and the cake was cut on next Sunday : %s.%n", javaBirthday, nextSunday);
	}
}
```

57.3.1. Understanding Random class

00:15

The Random class in `java.util` package can be used to generate pseudo random numbers.

Click on the class Random to know more about the available methods.

Observe the below code and predict the output.

```
package q11316;
import java.util.*;
public class RandomDemo {
   public static void main(String[] args) {
      Random rand = new Random(10);
      for (int i = 0; i < 5; i++) {
         System.out.println(rand.nextInt(100));
      }
   }
}
```

- (selected) generates 5 random numbers below 100
- generates 10 random numbers below 100
- generates 5 random numbers below 10
- None of these

58.1.1. Understanding multitasking and multithreading

00:26

The word Multitasking is applied to the operating system performing multiple tasks simultaneously. These tasks are called processes.

A process is a running instance of a program code, which uses resources like Memory, CPU cycles etc.

While you are reading this text, you already have many tasks (processes) being simultaneously run by your OS (operating system) like the browser instance, clock.. etc.

When we execute a Java class, a JVM (Java Virtual Machine) instance is created and launched as a separate program, which in turn executes the code in our Java class.

Java language has support for multithreading. Meaning we can write code which will create multiple threads. A thread is an independent execution flow that can run simultaneously along with its parent which launched it.

Click on the Live Demo button to see live threads in action. Make sure to fiddle around with pause and resume options.

The way processes are scheduled and managed by the operating system, threads that are created and run in a JVM have a one-to-one mapping to a corresponding OS threads (called native threads). For more details click here and read the **Threading Model** section..

In Java, threads have a priority which can be set. When a new thread is created, by default it receives the priority of its parent thread. This priority is used in thread scheduling.

Individual processes in an operating system have their own memory space. While the threads launched in a JVM, share the same context and the memory space used by the JVM. For this reason threads are also considered light-weight when compared to processes. though such a statement is debatable.

- (●) Multithreading is the execution of multiple threads within a single task, whereas multitasking is the execution of multiple tasks simultaneously.
- ( ) Multithreading is the execution of multiple tasks simultaneously, whereas multitasking is the execution of multiple threads within a single task.und tasks.
- ( ) Multithreading and multitasking are the same concepts in Java.
- ( ) Multithreading is used for CPU-bound tasks, whereas multitasking is used for I/O-bound tasks

58.2.1. Understanding Thread states

05:55

In Java a thread can be created and executed in two ways. One is by writing a class which extends the Thread class and calling the start() method.

```
Approach #1
public class MyClass extends Thread {
    public void run() {
   ...
    }
}
execute the thread by calling:
new MyThread().start();
```

The second approach is by writing a class which implements the Runnable interface. Then we create a Thread object by passing an instance of our class into the Thread's constructor and call the start() method.

```
Approach #2
public class MyClass implements Runnable {
    public void run() {
   ...
    }
}
execute the thread by calling:
MyClass mc =  new MyClass();
new Thread(mc).start();
```

Among the two, the **second approach** which implements a `Runnable` interface is recommended.

Click on the Single Thread Demo button to see a live thread in action. Make sure to fiddle around with pause and resume options.

Sample Test Cases +

Explorer

SleepDe...

Submit

```
package q36049;
class MyThread extends Thread {
    private int threadNumber;

    public MyThread(int threadNumber) {
        this.threadNumber = threadNumber;
    }
    public void run() {
        System.out.println("Thread: " + threadNumber);
    }}
class MyRunnable implements Runnable {
    private int mainNumber;

    public MyRunnable(int mainNumber) {
        this.mainNumber = mainNumber;
    }
    public void run() {
        System.out.println("Main: " + mainNumber);
    }}
public class SleepDemo {
    public static void main(String[] args) {
        // Approach #1: Using Thread class
        for (int i = 1; i <= 5; i++) {
            new MyThread(i).start();
        }
        // Approach #2: Using Runnable interface
        for (int i = 1; i <= 5; i++) {
            MyRunnable myRunnable = new MyRunnable(i);
            new Thread(myRunnable).start();
    }}}
```

Plots

58.2.2. Understanding Thread states

02:46

A thread can be in one of the below mentioned states:

1. NEW - when a thread is just created and is not started yet (meaning the start() method is not yet called on it).
2. RUNNABLE - when the start method is called and the thread is executing the code in run() method.
3. BLOCKED - when a thread is unable to proceed with execution because it is waiting for a monitor lock (we will learn more about locks later.)
4. WAITING - when a thread is waiting indefinitely for another thread to perform a particular action.
5. TIMED_WAITING - when a thread that is waiting for another thread to perform an action for a specified waiting time, after which it will resume.
6. TERMINATED - when a thread finishes its execution.

Note that when a new Thread is created it does not start automatically. At that moment it is in the **NEW** state. And after a thread's state changes to **TERMINATED**, it cannot be started again.

Note: We should never use the `stop()` method provided in the **Thread** class as it is deprecated and can lead to unexpected results. Instead we should manually write the code to stop a running thread.

Click on the Stop Thread Demo button to learn how to use a simple boolean flag to manually implement **stop()** feature in a thread.

Click on the Stop Feature - Multiple Threads Demo button to understand the working of **stop()** method in multiple threads.

Observe the code and the output provided and complete the code that demonstrates the states of a thread.

We should always remember that the `main` method is also executed in a thread by the JVM.

Sample Test Cases +

Explorer

SimpleTh...

Submit

Plots

```
        System.out.println("After start() method call t1.getState() = " + t1.getState());
        System.out.println("After start() method call t1.isAlive() = " + t1.isAlive());
        t1.join();
        System.out.println("After t1 has terminated t1.getState() = " + t1.getState());
        System.out.println("After t1 has terminated t1.isAlive() = " + t1.isAlive());
    }
}
class Counter implements Runnable {
    private String name;
    public Counter(String name) {
        this.name = name;
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(name + " : " + i);

            try {
                // Introduce some delay to simulate work
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

59.1.1. Understanding Java thread model

13:03

Threads in Java can be assigned priorities. The priority of a thread determines how it will behave with regards to other threads. For example if a low priority thread is hogging too much of CPU time and is not voluntarily surrendering CPU by sleeping or going into a wait, the system could pre-empt it and transfer the control to a thread with higher priority.

There can arise scenarios where multiple threads may want to access a shared resource. For example, let us consider two threads trying to add elements into an ArrayList. It is very likely that during the add operation call from a thread, while the ArrayList's internal data structure is being modified, the second thread can also invoke the add method. This could result in possible data corruption or even worse can mess up the internal state of the ArrayList.

Whenever a common resource is manipulated by multiple threads, it is important to endure that the manipulations happen in isolation of each other. Mean while one thread is performing the manipulation on the shared resource, the other threads are made to wait until the former completes. This is called **synchronization**.

Java allows us to write code which can build such synchronization between multiple threads by using the `synchronized` keyword.

In legacy collection classes like Vector, Stack and Hashtable all methods were **synchronized**.

A method is called synchronized when the `synchronized` keyword is used in the method declaration statement as shown below:

```
// Below code is from isEmpty() method in java.util.Vector class:
public synchronized boolean isEmpty() {
    return elementCount == 0;
}
```

Sample Test Cases +

Explorer

SimpleTh...

Submit

Plots

```
package q11343;
import java.util.*;
public class SimpleThreadDemo3 {
    public static void main(String[] args) throws InterruptedException {
        List<String> entriesList = Collections.synchronizedList(new ArrayList<String>());
        Counter c1 = new Counter("Ganga", entriesList);

        Counter c2 = new Counter("Yamuna", entriesList);
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
        t1.start();
        System.out.println("started t1 thread");
        t1.join();


        System.out.println("t1 has completed. t1.isAlive() = " + t1.isAlive());
        System.out.println("At the end entriesList contains ");
        for (String entry:entriesList) {
            System.out.println(entry);
        }
    }
}
class Counter implements Runnable {
    private String name;
    private List<String> entriesList;
    public Counter(String name, List<String> entriesList) {
        this.name = name;
        this.entriesList = entriesList;
```

59.1.2. Select all correct answers 00:14

Given:

```
class SequenceGenerator {
    synchronized void sequence(long n) {
        for (int i = 1; i < 3; i++)
            System.out.print(n + "-" + i + " ");
    }
}

public class Tester implements Runnable {
    static SequenceGenerator sg = new SequenceGenerator ();

    public static void main(String[] args) {
        new Thread(new Tester()).start();
        new Thread(new Tester()).start();
    }

    public void run() {
        sg.sequence(Thread.currentThread().getId());
    }
}
```

Which of the following statements are true?

- [ ] The output could be 5-1 6-1 6-2 5-2
- [x] The output could be 6-1 6-2 5-1 5-2
- [ ] The output could be 6-1 5-2 6-2 5-1
- [ ] The output could be 6-1 6-2 5-1 7-1

59.1.3. Understanding wait and notify

00:27

Let us consider a scenario where we have two threads, one thread downloads the files from the Internet and the other thread reads those downloaded files and extracts their content. In such a scenario, until the first thread has finished downloading the file completely from the Internet the second thread should not start reading it.

Java provides a way to solve such a problem by providing a means for the first thread to signal the second thread through **wait** and **notify** mechanism.

The Object class which is the root call of all classes in Java, has the below to facilitate **wait** and **notify** between threads:

1. wait() - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object.
2. wait(long timeoutInMilliSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
3. wait(long timeoutInMilliSeconds, int additionalNanoSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
4. notify() - it causes one of the threads waiting on this object's intrinsic lock to wake up.
5. notifyAll() - it causes all the threads waiting on this object's intrinsic lock to wake up, while only one thread gets a chance to obtain the lock to proceed.

If any thread is waiting using any of the above mentioned **wait** methods, it will come out of its wait, if an Thread.interrupt() is called on that thread.

The **wait** or an **notify** call can be made on a object only after obtaining the **intrinsic lock** (also called **monitor**) on that object by synchronizing over it.

If we try to call **wait** or **notify** on a object without synchronizing on that object an

Sample Test Cases +

Explorer

WaitNotif...

Submit

```java
package q11345;
public class WaitNotifyDemo {
    public static void main(String[] args) throws InterruptedException {
        Object sharedLock = new Object();
        Waiter waiter = new Waiter(sharedLock);
        Notifier notifier = new Notifier(sharedLock);
        Thread t1 = new Thread(waiter);
        Thread t2 = new Thread(notifier);
        t1.start();
        t2.start();
        t2.join();
        t1.join();
    }}
class Waiter implements Runnable {
    private Object sharedLock;
    public Waiter(Object sharedLock) {
        this.sharedLock = sharedLock;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
        System.out.println("Waiter : " + i);
            if (i == 5) {        System.out.println("Waiter will wait now until notified by Notifier");
            synchronized (sharedLock) {
            try {
                sharedLock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
    }}
```

Plots

< Prev Reset Submit Next >

59.1.3. Understanding wait and notify

00:27

Let us consider a scenario where we have two threads, one thread downloads the files from the Internet and the other thread reads those downloaded files and extracts their content. In such a scenario, until the first thread has finished downloading the file completely from the Internet the second thread should not start reading it.

Java provides a way to solve such a problem by providing a means for the first thread to signal the second thread through **wait** and **notify** mechanism.

The Object class which is the root call of all classes in Java, has the below to facilitate **wait** and **notify** between threads:

1. wait() - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object.
2. wait(long timeoutInMilliSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
3. wait(long timeoutInMilliSeconds, int additionalNanoSeconds) - it causes the current thread to wait until another thread invokes the notify() or notifyAll() on this object or the specified timeout happend.
4. notify() - it causes one of the threads waiting on this object's intrinsic lock to wake up.
5. notifyAll() - it causes all the threads waiting on this object's intrinsic lock to wake up, while only one thread gets a chance to obtain the lock to proceed.

If any thread is waiting using any of the above mentioned **wait** methods, it will come out of its wait, if an Thread.interrupt() is called on that thread.

The **wait** or an **notify** call can be made on a object only after obtaining the **intrinsic lock** (also called **monitor**) on that object by synchronizing over it.

If we try to call **wait** or **notify** on a object without synchronizing on that object an

Sample Test Cases +

Explorer

WaitNotif...

Submit

```
34        for (int i = 0; i < 10; i++) {
35            System.out.println("Waiter : " + i);
36            if (i == 5) {
              System.out.println("Waiter will wait now until notified by Notifier");
37            synchronized (sharedLock) {
38                try {
39                    sharedLock.wait();
40                } catch (InterruptedException e) {
41                    e.printStackTrace();
42                }}
43            System.out.println("Waiter has come out of wait.");
44        }}}}
45    class Notifier implements Runnable {
46        private Object sharedLock;
47        public Notifier(Object sharedLock) {
48            this.sharedLock = sharedLock;
49        }
50        public void run() {
51            System.out.println("Notifier is busy taking a nap to 7 secs.");
52            try {
53                Thread.sleep(200);
54            } catch (InterruptedException e) {
55                e.printStackTrace();
56            }
57            System.out.println("Hurray! Notifier is awake and is about to call notify().");
58            synchronized (sharedLock) {
59                sharedLock.notify();
60            }}}
```

Plots

< Prev Reset Submit Next >

59.1.4. Select the correct answers 01:02

Read the code below:

```
void waitForSignal() throws InterruptedException {
    Object obj = new Object();
    synchronized (Thread.currentThread()) {
        obj.wait();
        obj.notify();
     }
}
```

Which statement is true?

- [ ] Executing the code results in InterruptedException.
- [x] The code throws IllegalMonitorStateException.
- [ ] The code throws a TimeoutException after ten minutes.
- [ ] Reversing the order of obj.wait() and obj.notify() might cause this method to complete normally.
- [ ] A call to notify() or notifyAll() from another thread might cause this method to complete normally.
- [ ] This code does NOT compile unless "obj.wait()" is replaced with "((Thread)obj).wait()".

60.1.1. Understanding Thread and Runnable

21:48

When a Java program is launched, JVM uses the `main` method in a class as the starting point for code execution. Similarly, the `run` is the starting point for a **Thread**.

It is important to note that the `run()` should not be called directly. If called directly it will executed as any other normal method in the context of the current thread sequentially.

When we want the code in the `run()` method to be executed in a separate thread simultaneously (asynchronously), we hand over the runnable instance (which contains the `run()` method) to a thread object and ask the thread object to start executing. The call to the `start()` method on the thread object triggers the execution of the code in the `run()` in that thread's context.

Java also provides executors which generally hold a pool of threads to execute runnable instances. These are recommended in large-scale applications for efficient management of threads. Different types of executors can be created by calling the static methods available in the utility class Executors which is present in the java.util.concurrent package.

Fill the missing code in the given code to learn the difference between calling `run()` method directly and executing it in a thread.

You will notice in the output that the thread executing the `main` method itself is busy executing the `c1.run();`. And until it complete the `run()` on `c1`, `main` method does not even proceed to create and start threads `t1` and `t2`. However, once the threads `t1` and `t2` are started, from the output you notice that the CPU time is sliced between both the threads and both of them are executing simultaneously.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Runnable...

Submit

Plots

```
package q36053;
public class RunnableDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c1 = new Counter("Ganga");
        Counter c2 = new Counter("Yamuna");
        Counter c3 = new Counter("Narmada");
        c1.run();
        Thread t1 = new Thread(c2);
        Thread t2 = new Thread(c3);
            t1.start();
        System.out.println("called t1.start()");
        t2.start();
        System.out.println("called t2.start()");
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("t1 has completed. t1.isAlive() = " + t1.isAlive());
        System.out.println("t2 has completed. t2.isAlive() = " + t2.isAlive());
    }
}

class Counter implements Runnable {
    private String name;

```

< Prev Reset Submit Next >

60.1.1. Understanding Thread and Runnable

21:48

When a Java program is launched, JVM uses the `main` method in a class as the starting point for code execution. Similarly, the `run` is the starting point for a **Thread**.

It is important to note that the `run()` should not be called directly. If called directly it will executed as any other normal method in the context of the current thread sequentially.

When we want the code in the `run()` method to be executed in a separate thread simultaneously (asynchronously), we hand over the runnable instance (which contains the `run()` method) to a thread object and ask the thread object to start executing. The call to the `start()` method on the thread object triggers the execution of the code in the `run()` in that thread's context.

Java also provides executors which generally hold a pool of threads to execute runnable instances. These are recommended in large-scale applications for efficient management of threads. Different types of executors can be created by calling the static methods available in the utility class Executors which is present in the java.util.concurrent package.

Fill the missing code in the given code to learn the difference between calling `run()` method directly and executing it in a thread.

You will notice in the output that the thread executing the `main` method itself is busy executing the `c1.run();`. And until it complete the `run()` on `c1`, `main` method does not even proceed to create and start threads `t1` and `t2`. However, once the threads `t1` and `t2` are started, from the output you notice that the CPU time is sliced between both the threads and both of them are executing simultaneously.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Runnable...

Submit

Plots

```
20
21          System.out.println("t1 has completed. t1.isAlive() = " + t1.isAlive());
22          System.out.println("t2 has completed. t2.isAlive() = " + t2.isAlive());
23      }
24  }
25
26  class Counter implements Runnable {
27      private String name;
28
29      public Counter(String name) {
30          this.name = name;
31      }
32
33      public void run() {
34          for (int i = 0; i < 3; i++) {
35              System.out.println(name + " : " + i);
36
37              try {
38                  Thread.sleep(500);
39              } catch (InterruptedException e) {
40                  e.printStackTrace();
41              }
42          }
43      }
44  }
```

Prev Reset Submit Next

60.1.2. Select the correct answers 01:17

Read the following code and select **all** the correct outputs:

```
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("In run() method ");
        throw new RuntimeException("some problem");
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
        System.out.println("End of main() method");
    }
}
```

- [ ] In run() method
- [ ] End of main() method
- [ ] some problem
- [x] In run() method
End of main() method
java.lang.RuntimeException: some problem
- [ ] In run() method
java.lang.RuntimeException: some problem
- [x] In run() method
java.lang.RuntimeException: some problem
End of main() method
- [x] End of main() method
In run() method
java.lang.RuntimeException: some problem
- [ ] java.lang.RuntimeException: some problem

60.1.3. Write a Java program demonstrating the usage of Threads

00:25

Write a Java program that uses three threads to perform the below actions:

1. First thread should print "Good morning" for every 1 second for 2 times
2. Second thread should print "Hello" for every 1 seconds for 2 times
3. Third thread should print "Welcome" for every 3 seconds for 1 times

Write appropriate **constructor** in the `Printer` class which implements `Runnable` interface to take three arguments : **message**, **delay** and `count` of types **String**, **int** and **int** respectively.

Write code in the `Printer.run()` method to print the **message** with appropriate **delay** and for number of times mentioned in **count**.

Write a class called `ThreadDemo` with the `main()` method which instantiates and executes three instances of the above mentioned `Printer` class as threads to produce the desired output.

[**Note:** If you want to sleep for **2** seconds you should call `Thread.sleep(2000);` as the `Thread.sleep(...)` method takes milliseconds as argument.]

Note: Please don't change the package name.

Sample Test Cases +

Explorer

ThreadDe...

Submit

Plots

```java
package q11349;
public class ThreadDemo {
	public static void main(String[] args) throws Exception {
		Thread t1 = new Thread(new Printer("Good morning", 1, 2));
		Thread t2 = new Thread(new Printer("Hello", 1, 2));
		Thread t3 = new Thread(new Printer("Welcome", 3, 1));
		t1.start();
		t2.start();
		t3.start();
		t1.join();
		t2.join();
		t3.join();
		System.out.println("All the three threads t1, t2 and t3 have completed execution.");
	}
}
class Printer implements Runnable {
	private String message;

	private int delay,count;

	public Printer(String message,int delay,int count) {

		this.message = message;

		this.delay=delay;

		this.count=count;
```

< Prev Reset Submit Next >

60.1.3. Write a Java program demonstrating the usage of Threads

00:25

Write a Java program that uses three threads to perform the below actions:

1. First thread should print "Good morning" for every 1 second for 2 times
2. Second thread should print "Hello" for every 1 seconds for 2 times
3. Third thread should print "Welcome" for every 3 seconds for 1 times

Write appropriate **constructor** in the `Printer` class which implements `Runnable` interface to take three arguments : **message**, **delay** and `count` of types **String**, **int** and **int** respectively.

Write code in the `Printer.run()` method to print the **message** with appropriate **delay** and for number of times mentioned in **count**.

Write a class called `ThreadDemo` with the `main()` method which instantiates and executes three instances of the above mentioned `Printer` class as threads to produce the desired output.

[**Note:** If you want to sleep for **2** seconds you should call `Thread.sleep(2000);` as the `Thread.sleep(...)` method takes milliseconds as argument.]

Note: Please don't change the package name.

Sample Test Cases +

Explorer

ThreadDe...

Submit

Plots

```
}
class Printer implements Runnable {
    private String message;

    private int delay,count;

    public Printer(String message,int delay,int count) {

        this.message = message;

        this.delay=delay;

        this.count=count;

    }

    public void run() {
        //long value =delay*1000;

        for (int i = 0; i < count; i++) {

            //Thread.sleep(value);

            System.out.println(message);

        }

    }


}
```

< Prev Reset Submit Next >

60.1.4. Write a Java program to illustrates isAlive() and join()

00:34

Write a Java program to demonstrate the usage of `isAlive()` and `join()` methods in threads.

Write a class `JoinThreadDemo` with a `main()` method that creates and executes two instances of `Counter` class which implements `Runnable` interface.

Let the `Counter` class take a **String** argument **name** and let its `run()` method print that message for **10** times along with the current count as given below:

```
System.out.println(name + " : " + i);
```

The `JoinThreadDemo.main()` method should perform the below tasks in the given order :

1. Create the first instance of thread as `t1` with an instance of `Counter` class using **"Spain"** as the argument.
2. Create the second instance of thread as `t2` with an instance of `Counter` class using **"UAE"** as the argument.
3. Print the `isAlive()` status of `t1` as : **"t1 before start t1.isAlive() : " + t1.isAlive()**.
4. Print the `isAlive()` status of `t1` as : **"t2 before start t2.isAlive() : " + t2.isAlive()**.
5. Start `t1` and `t2` threads respectively.
6. Print a message to the console as : **"started t1 and t2 threads"**.
7. Print the `isAlive()` status of `t1` as : **"t1 after start t1.isAlive() : " + t1.isAlive()**.
8. Invoke the `join()` method on `t2`.
9. Print the `isAlive()` status of `t1` as : **"t2 after start t2.isAlive() : " + t2.isAlive()**.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

JoinThre...

Submit

```
package q11350;
public class JoinThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 =new Thread(new Counter("Spain") );
        Thread t2 =new Thread (new Counter("UAE")) ;
        System.out.println("t1 before start t1.isAlive() : "+t1.isAlive());
        System.out.println("t2 before start t2.isAlive() : "+t2.isAlive());
        t1.start();
        t2.start();
        System.out.println("started t1 and t2 threads");
        System.out.println("t1 after start t1.isAlive() : "+t1.isAlive());
        t2.join();
        System.out.println("t2 after start t2.isAlive() : "+t2.isAlive());
    }}
class Counter implements Runnable {
    private String name;
    public Counter(String name) {
        this.name = name;
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
        System.out.println(name + " : " + i);
        }}}
```

Plots

< Prev Reset Submit Next >

60.1.5. Write a Java program to illustrate Daemon Threads

00:18

Daemon thread is a low priority thread (in the context of **JVM**) that runs in background to perform tasks such as garbage collection etc., they do not prevent the **JVM** from exiting when all the user threads finish their execution.

JVM terminates itself when all user threads finish their execution, even while Daemon threads are running.

The code `Thread.currentThread().isDaemon()` will return `true`, if the current thread **is a daemon** thread and `false` if it **is not a daemon** thread.

Write a Java program to illustrate **daemon threads**.

Write a class `DeamonThreadDemo` which extends the `Thread` class. Override its **run()** method to check whether the current thread is either **daemon** or **user thread** and print **"This is daemon thread"** and **"This is not a daemon thread"** respectively.

Write the **main()** method in the class `DeamonThreadDemo`, which create three instances of class `DaemonThreadDemo` as `t1`, `t2` and `t3` and perform the below tasks in the given order:

1. invoke the `setDaemon()` method on `t1` instance and pass **true** as the argument to set `t1` as a daemon thread.
2. Invoke `start()` method on `t1`, `t2` and `t3` respectively.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

DaemonT...

Submit

Plots

```java
package q11351;
public class DaemonThreadDemo extends Thread {
    public void run() {
        if (Thread.currentThread().isDaemon() ) {
            System.out.println("This is daemon thread");
        } else {
            System.out.println("This is not a daemon thread");
        }
    }
    public static void main(String[] args) {
        DaemonThreadDemo t1 = new DaemonThreadDemo();
        DaemonThreadDemo t2 = new DaemonThreadDemo();
        DaemonThreadDemo t3 = new DaemonThreadDemo();

        t1.setDaemon(true);
        t1.start();t2.start();t3.start();


    }
}
```

< Prev Reset Submit Next >