

## **Day 3- Facilitation Guide**

### **Flow Control and Inheritance**

#### **Index**

- I. Flow Control
- II. Inheritance

#### **For (1.5 hrs) ILT**

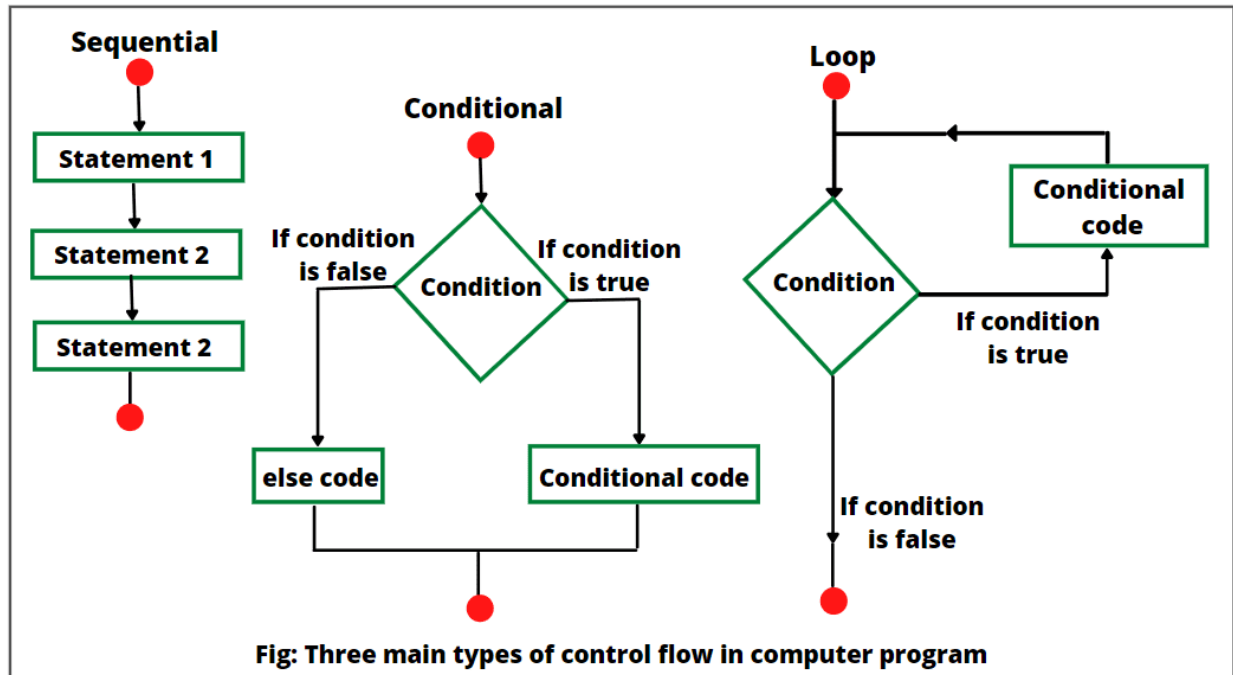
In the previous session we learnt about the OOPs concepts. We defined the structure of a Java class and explained in detail about instantiation of a Java class. We also learnt about methods and constructor. We learnt about operators and their usage. You should now have a clear idea about JavaBeans and the advantages of using JavaBeans.

In this session we will explore the flow controls. We will also learn in detail about inheritance with proper examples.

**Before proceeding further let us summarize some of the important lessons we learned so far:**

- Java is an object-oriented programming language.
- Java uses classes to encapsulate the data members and methods.
- All public classes, filenames must be the same as the class names. File extension of Java source file should be ".java".
- A main() method with signature public static void main(String[] args) is required to execute a Java code.
- The "new" operator along with the constructor name is used to create an instance of a class. Instance of a class is called an object.
- Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which encapsulates the medical substance. We can create a fully encapsulated class in Java by making all the data members of the class private.
- Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while \* is an operator used for multiplication.

**Note:** Before transitioning to the new topic, the trainer will inquire whether the students are familiar with flow control and about the synergy between operators and flow control in programming, exploring how they can be combined to create dynamic and efficient applications.



## I. Flow Control

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

### 1. Decision Making statements

- if statements
- switch statement

### 2. Loop statements

- do while loop
- while loop
- for loop
- for-each loop

### 3. Jump statements

- break statement
- continue statement

In this section, you will explore the various control flow statements offered by the Java programming language, including decision-making (if-then, if-then-else, switch), looping (for, while, do-while), and branching (break, continue, return) statements.

**Question:** The trainer will prompt students to explore real-world applications where decision-making constructs (e.g., switch statements and if-else blocks) and looping constructs (such as for, while, and do-while loops) are utilized.

## If-Else

### The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes() {  
    // the "if" clause: bicycle must be moving  
    if (isMoving){  
        // the "then" clause: decrease current speed  
        currentSpeed--;  
    }  
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {  
    // same as above, but without braces  
    if (isMoving)  
        currentSpeed--;  
}
```

Deciding when to omit the braces is a matter of personal choice. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

### **The if-then-else Statement**

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the `applyBrakes()` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {  
    if (isMoving) {  
        currentSpeed--;  
    } else {  
        System.err.println("The bicycle has already stopped!");  
    }  
}
```

The following program, `IfElseDemo`, assigns a grade based on the value of a test score. An A for a score of 90% or above, a B for a score of 80% or above, and so on.

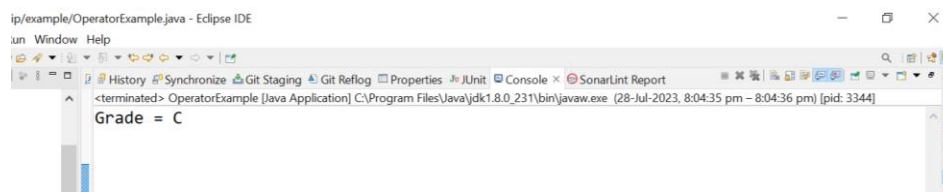
```
class IfElseDemo {  
    public static void main(String[] args) {  
  
        int testscore = 76;  
        char grade;  
  
        if (testscore >= 90) {
```

```

        grade = 'A';
    } else if (testscore >= 80) {
        grade = 'B';
    } else if (testscore >= 70) {
        grade = 'C';
    } else if (testscore >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    System.out.println("Grade = " + grade);
}
}

```

The output from the program is:



You may have noticed that the value of testscore can satisfy more than one expression in the compound statement:  $76 \geq 70$  and  $76 \geq 60$ . However, once a condition is satisfied, the appropriate statements are executed ( $\text{grade} = \text{'C'}$ ;) and the remaining conditions are not evaluated.

### Knowledge check:

1.

```
int x = 10;
```

```
int y = 5;
```

```

if (x > y) {
    System.out.println("x is greater than y");
} else {
    System.out.println("y is greater than or equal to x");
}

```

**What will be the output of this code?**

- A) x is greater than y
- B) y is greater than or equal to x
- C) x is greater than or equal to y
- D) None of the above

**Ans:** x is greater than y

**2.**

```
int age = 18;
```

```
if (age < 18) {  
    System.out.println("You are a minor.");  
} else if (age == 18) {  
    System.out.println("Congratulations! You just became an adult.");  
} else {  
    System.out.println("You are an adult.");  
}
```

**What will be the output of this code if age = 18?**

- A) You are a minor.
- B) Congratulations! You just became an adult.
- C) You are an adult.
- D) None of the above

**Ans:** Congratulations! You just became an adult.

**3.**

```
int num = 7;
```

```
if (num % 2 == 0) {  
    System.out.println("The number is even.");  
} else {  
    System.out.println("The number is odd.");  
}
```

**What will be the output of this code if num = 7?**

- A) The number is even.
- B) The number is odd.

- C) The number is prime.
- D) None of the above

**Ans:**The number is odd.

## **Switch Statements**

In Java, switch is a control flow statement used for decision-making. It allows you to select one of many code blocks to be executed based on the value of an expression. The expression in a switch statement is evaluated once, and its value is then compared with the values of each case. If there is a match, the corresponding block of code associated with that case is executed.

A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types, the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer.

**The basic syntax of a switch statement in Java is as follows:**

```
switch (expression) {  
  
    case value1:  
  
        // Code block to be executed if 'expression' equals 'value1'  
  
        break;  
  
    case value2:  
  
        // Code block to be executed if 'expression' equals 'value2'  
  
        break;  
  
    // More cases can be added here  
  
    default:  
  
        // Code block to be executed if no case matches the 'expression'  
  
        break;  
  
}
```

### Here's how it works:

- The expression is usually a variable or a constant that is evaluated once.
- The case labels are the possible values that the expression might have.
- If the value of the expression matches any of the case labels, the corresponding block of code under that case is executed.
- If there is no match between the expression and any of the case labels, the code under the default label (optional) is executed.
- The break statement is used to exit the switch block after a case has been matched and executed. Without a break, the execution will "fall through" to the next case, and all subsequent cases will be executed, regardless of whether their values match the expression.

```
package com.anudip.example;
```

```
public class SwitchCaseEx {  
  
    public static void main(String[] args) {  
        int dayOfWeek = 2;  
        String dayName;  
  
        switch (dayOfWeek) {  
            case 1:  
                dayName = "Sunday";  
                break;  
            case 2:  
                dayName = "Monday";  
                break;  
            case 3:  
                dayName = "Tuesday";  
                break;  
            case 4:  
                dayName = "Wednesday";  
                break;  
            case 5:  
                dayName = "Thursday";  
                break;  
            case 6:  
                dayName = "Friday";  
                break;  
            case 7:  
                dayName = "Saturday";  
        }  
    }  
}
```



```

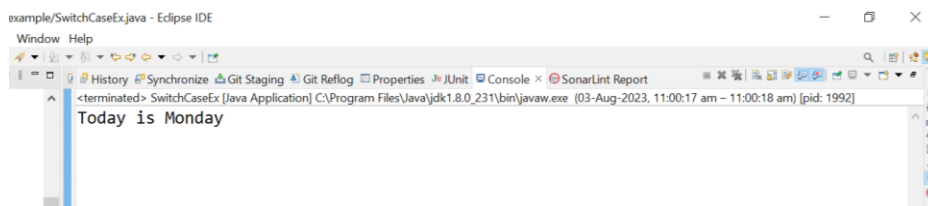
        break;
    default:
        dayName = "Invalid day";
        break;
    }

    System.out.println("Today is " + dayName);

}
}

```

### **Output:**



## **Loop**

Looping Constructs in Java are statements that allow a set of instructions to be performed repeatedly as long as a specified condition remains true. Java has three types of loops i.e. the for loop, the while loop, and the do-while loop.

### **The while and do-while Statements**

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```

while (expression) {

    statement(s)

}

```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement(s)* in the

while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {  
  
    public static void main(String[] args){  
  
        int count = 1;  
  
        while (count < 11) {  
  
            System.out.println("Count is: " + count);  
  
            count++;  
  
        }  
  
    }  
  
}
```



You can implement an infinite loop using the while statement as follows:

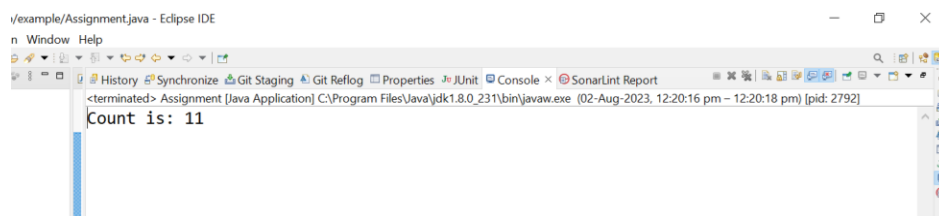
```
while (true){  
  
    // your code goes here  
  
}
```

The Java programming language also provides a **do-while** statement, which can be expressed as follows:

```
do {  
    statement(s)  
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {  
    public static void main(String[] args){  
        int count = 11;  
        do {  
            System.out.println("Count is: " + count);  
            count++;  
        } while (count < 11);  
    }  
}
```



In the above example, the count value is 11, and it still gets printed even though the condition should have been <11, as the check is performed at the end.

**Food for thought..**

*Are you fascinated by social media platforms and their dynamic content? How might looping in Java be used to continuously update feeds, notifications, and handle real-time interactions?*

## The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {  
    statement(s)  
}
```

**When using this version of the for statement, keep in mind that:**

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {  
    public static void main(String[] args){  
        for(int i=1; i<11; i++){  
            System.out.println("Count is: " + i);  
        }  
    }  
}
```

**The output of this program is:**



Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression.

The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
// infinite loop
```

```
for ( ; ; ) {
```

```
    // your code goes here
```

```
}
```

### **Food for thought..**

*The trainer can ask the students the following question to engage them in a discussion about the use of infinite loops:*

*"Have you ever wondered why someone might use an infinite loop in programming? What do you think could be some practical scenarios where an infinite loop could be beneficial or necessary?"*

**Here is the answer**

The infinite for loop in Java is used when you need to execute a block of code repeatedly, indefinitely, without an explicit exit condition. It is created by omitting the initialization, condition, and update expressions in the for loop syntax, resulting in a loop that continuously executes the specified code.

### **The infinite for loop can be useful in some specific scenarios:**

**Real-time Systems:** In real-time systems, you may need to continuously monitor and react to events or input from external sources. An infinite for loop can be used to keep the system running and handle events as they occur.

**Game Loops:** In game development, the game loop is responsible for updating the game state, rendering graphics, and handling user input continuously. An infinite for loop is often used to maintain the game loop's continuous execution.

However, it's important to exercise caution when using an infinite for loop because, without a proper exit condition, it can lead to an infinite loop, causing the program to become unresponsive or hang. In most cases, you should include an explicit exit mechanism within the loop or ensure that the loop can be broken out of based on certain conditions.

### **Enhanced For Loop:**

The for statement also has another form designed for iteration through Collections and arrays. This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
class EnhancedForDemo {  
  
    public static void main(String[] args){  
  
        int[] numbers =  
            {1,2,3,4,5,6,7,8,9,10};  
  
        for (int item : numbers) {
```

```

        System.out.println("Count is: " + item);
    }
}
}

```

In this example, the variable `item` holds the current value from the `numbers` array. The output from this program is the same as before:



```

Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10

```

We recommend using this form of the `for` statement instead of the general form whenever possible.

## Knowledge check...

***Now that we've covered different types of looping statements, it's time to test your understanding.***

***The trainer will present a series of knowledge check questions on Java looping concepts. After asking the questions, the trainer will lead the group in open discussions to delve deeper into each topic, encouraging active participation and sharing of insights among all attendees.***

1. What are the three types of loops in Java, and how do they differ in their use cases?
2. Explain the difference between the "while" loop and the "do-while" loop in Java. When would you use one over the other?
3. What are the advantages of using a "for" loop over a "while" loop or vice versa? When would you prefer one type of loop over the other in your code?

## Branching Statements

To gain a better understanding of the importance of Branching Statements in programming, let's explore their significance before diving into the topic.

Branching statements are required in Java (and other programming languages) to introduce conditional logic, enabling the program to make decisions and change its behavior based on specific conditions. Branching statements provide the ability to create different paths of execution within a program, making it more flexible and capable of handling various scenarios. Here are some key reasons why branching statements are essential in Java:

**User Interaction:** In many applications, users interact with the program, providing input or making choices. Branching statements enable the program to respond to user actions, validate input, and provide appropriate feedback or actions based on the user's choices.

**Loop Control:** Branching statements like "break" and "continue" are used to control loops. They allow the program to exit a loop prematurely or skip certain iterations based on specific conditions, optimizing the loop's execution and efficiency.

Overall, branching statements are an essential tool for creating efficient, interactive, and adaptive programs. They empower developers to design complex logic, handle different scenarios, and build applications that cater to various user needs and requirements. Without branching statements, programs would follow a rigid and linear path, limiting their usefulness and responsiveness.

**Now, let's delve into understanding Branching Statements.**

## **Branching Statements**

Branching statements are the statements used to jump the flow of execution from one part of a program to another. The branching statements are mostly used inside the control statements. Java has mainly three branching statements, i.e., continue, break, and return.

### **The break Statement**



The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following BreakDemo program:

```
class BreakDemo {  
  
    public static void main(String[] args) {  
  
        int[] arrayOfInts =  
            { 32, 87, 3, 589,  
              12, 1076, 2000,  
              8, 622, 127 };  
  
        int searchfor = 12;  
  
        int i;  
  
        boolean foundIt = false;  
  
        for (i = 0; i < arrayOfInts.length; i++) {  
            if (arrayOfInts[i] == searchfor) {  
                foundIt = true;  
  
                break;  
            }  
        }  
  
        if (foundIt) {  
            System.out.println("Found " + searchfor + " at index " + i);  
        } else {  
            System.out.println(searchfor + " not in the array");  
        }  
    }  
}
```

```
}  
  
}
```

This program searches for the number 12 in an array. The break statement, shown in bold font, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop.

**This program's output is:**



An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement.

The following program, BreakWithLabelDemo, is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

```
class BreakWithLabelDemo {  
  
    public static void main(String[] args) {  
  
        int[][] arrayOfInts = {  
            { 32, 87, 3, 589 },  
            { 12, 1076, 2000, 8 },  
            { 622, 127, 77, 955 }  
        };  
  
        int searchfor = 12;  
  
        int i;  
  
        int j = 0;
```

```
boolean foundIt = false;
```

```
search:
```

```
for (i = 0; i < arrayOfInts.length; i++) {
```

```
    for (j = 0; j < arrayOfInts[i].length;
```

```
        j++) {
```

```
            if (arrayOfInts[i][j] == searchfor) {
```

```
                foundIt = true;
```

```
                break search;
```

```
            }
```

```
        }
```

```
    }
```

```
if (foundIt) {
```

```
    System.out.println("Found " + searchfor + " at " + i + ", " + j);
```

```
} else {
```

```
    System.out.println(searchfor + " not in the array");
```

```
}
```

```
}
```

```
}
```

**This is the output of the program.**



The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow exits the labeled statement and goes to the next instructions.

### The continue Statement

The continue statement skips the current iteration of a for, while, or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.

The following program, ContinueDemo, steps through a String, counting the occurrences of the letter "p". If the current character is not a 'p', the continue statement skips the rest of the loop and proceeds to the next character. If it is a "p", the program increments the letter count.

#### Example:

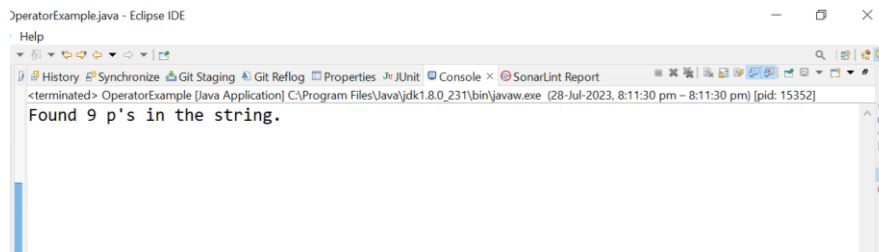
```
class ContinueDemo {  
  
    public static void main(String[] args) {  
  
        String searchMe = "peter piper picked a " + "peck of pickled peppers";  
  
        int max = searchMe.length();  
  
        int numPs = 0;  
  
        for (int i = 0; i < max; i++) {  
  
            // interested only in p's  
  
            if (searchMe.charAt(i) != 'p')
```

```
        continue;

    // process p's
    numPs++;
}

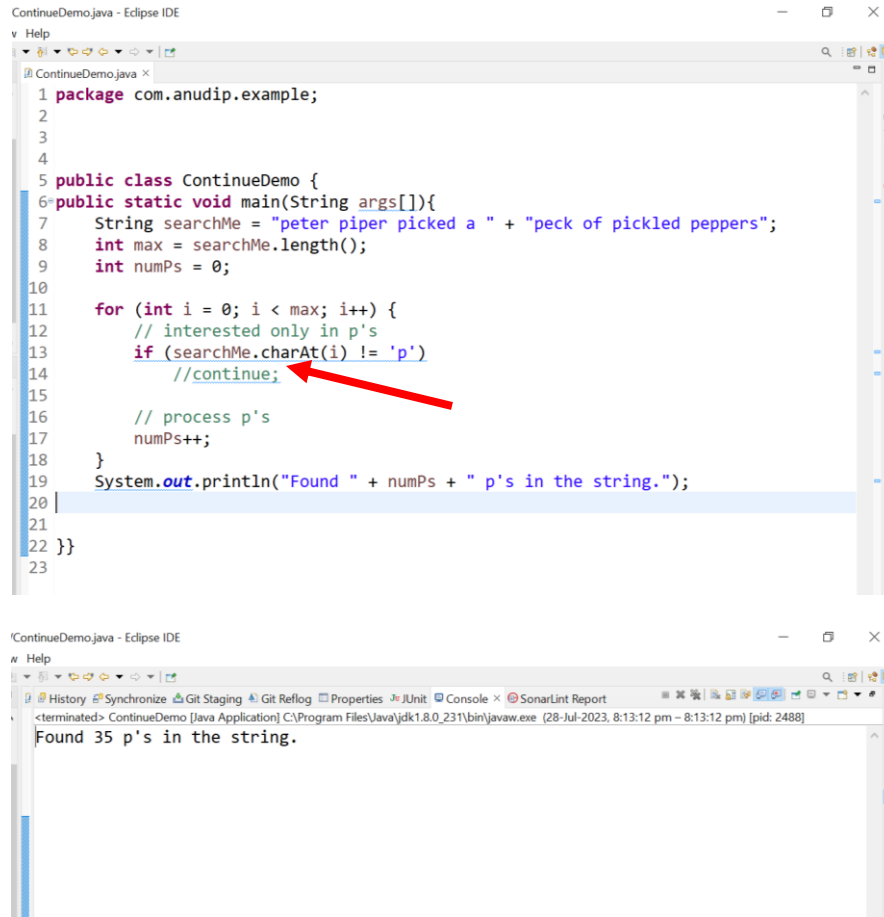
System.out.println("Found " + numPs + " p's in the string.");
}
}
```

Here is the output of this program:



To see this effect more clearly, **try removing the continue statement and recompiling**. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

**Here is the result:**



A labeled continue statement skips the current iteration of an outer loop marked with the given label.

The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, `ContinueWithLabelDemo`, uses the labeled form of `continue` to skip an iteration in the outer loop.

### Example:

```
class ContinueWithLabelDemo {

    public static void main(String[] args) {

        String searchMe = "Look for a substring in me";

        String substring = "sub";

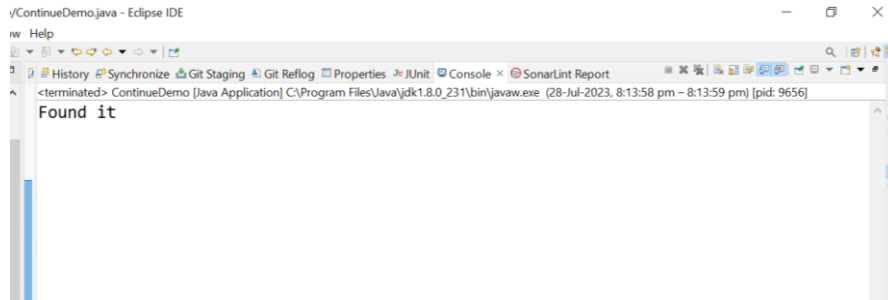
        boolean foundIt = false;
```

```
int max = searchMe.length() -  
    substring.length();
```

**test:**

```
for (int i = 0; i <= max; i++) {  
    int n = substring.length();  
    int j = i;  
    int k = 0;  
    while (n-- != 0) {  
        if (searchMe.charAt(j++) != substring.charAt(k++)) {  
            continue test;  
        }  
    }  
    foundIt = true;  
    break test;  
}  
System.out.println(foundIt ? "Found it" : "Didn't find it");  
}  
}
```

**Here is the output from this program.**



## The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.


```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value.

```
public class returnDemo {  
  
    //this method will calculate the factorial number  
    public int fact(int n)  
    {  
        //fact variable will be holding the factorial result  
        int fact=1;  
        //this loop is iterating from 1 to number  
        for(int i=1;i<=5;i++)  
        {  
            fact*=i; //fact=fact*i; //calculating factorial  
        }  
        return fact; //returning the result of factorial of specific number  
    }  
    public static void main(String args[]){  
  
        returnDemo demo=new returnDemo();  
        System.out.println("factorial of 5 is:"+ demo.fact(5));  
    }  
}
```



}}



The screenshot shows the Eclipse IDE interface. The title bar reads 'urnDemo.java - Eclipse IDE'. The console window at the bottom displays the following text: '<terminated> returnDemo [Java Application] C:\Program Files\Java\jdk1.8.0\_231\bin\javaw.exe (28-Jul-2023, 8:27:09 pm - 8:27:09 pm) [pid: 2200]' followed by 'factorial of 5 is:120'.

## Knowledge check...

***Now that we've covered branching statements, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.***

**1.What does the break statement do in Java?**

- A) Terminates the current loop iteration and continues with the next iteration.
- B) Exits the loop entirely, regardless of any remaining iterations.
- C) Terminates the entire program.
- D) Breaks the current method execution and returns a value.

Ans-B) Exits the loop entirely, regardless of any remaining iterations.

**2.Which statement is used to terminate the execution of a method and return a value from the method in Java?**

- A) stop
- B) end
- C) break
- D) return

ANs-D) return

**3.Can the return statement be used to exit a loop in Java?**

- A) Yes, return can exit a loop.
- B) No, return cannot exit a loop.

Ans-B) No, return cannot exit a loop. The return statement is used to exit a method and return a value, not a loop.

#### **4.What does the continue statement do in Java?**

- A) Exits the loop entirely, regardless of any remaining iterations.
- B) Skips the current loop iteration and continues with the next iteration.
- C) Terminates the entire program.
- D) Continues with the current loop iteration and ignores the rest of the code.

Ans-B) Skips the current loop iteration and continues with the next iteration.

***At the end of the quiz, the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.***

**Note:** Before moving to the next concept, the trainer can increase curiosity in students for upcoming concepts so that it can stimulate interest and encourage exploration.

#### **Here are some Tips:**

- Ask them what is the meaning of inheritance in the real-world and cite a real-world example such as a child inheriting characteristics and behavior from its parents.
- Use the animal kingdom as an analogy to explain inheritance. Show how various animals share common characteristics within a base "Animal" class but have specialized features in their derived classes.
- Present real-life examples where inheritance is employed, such as programming frameworks, libraries, or popular Java APIs. Discuss how understanding inheritance can help students comprehend these systems..
- Now, let's explore further examples of inheritance. Trainer would like to encourage all students to participate and share additional instances where inheritance can be applied in various scenarios.

## **II. Inheritance**

We had introduced inheritance in the previous session. Inheritance is a concept in which one object acquires/inherits another object's properties, and inheritance also supports hierarchical classification. The idea behind this is that we can create new classes built on existing classes, i.e., when you inherit from an existing class, we can reuse methods and fields of the parent class. Inheritance represents the parent-child relationship.

For example, a whale is a part of the classification of marine animals, which is part of class mammal, which is under that class of animal.

We use hierarchical classification, i.e., top-down classification. If we want to describe a more specific class of animals such as mammals, they would have more specific attributes such as teeth; cold-blooded, warm-blooded, etc. This comes under the subclass of animals whereas animals come under the superclass.

The subclass is a class which inherits properties of the superclass. This is also called a derived class. A superclass is a base class or parent class from which a subclass inherits properties.

### **Why Do We Need Inheritance in Java?**

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Runtime Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

### **Important Terminologies Used in Java Inheritance**

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.

- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class/Child Class/Derived class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

### How to achieve inheritance in Java?

The extends keyword is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

### Syntax :

```
class derived-class extends base-class
{
    //methods and fields
}
```

### Example:

```
package com.anudip.example;
    // Java program to illustrate the
    // concept of inheritance

    // base class
    class Bicycle {
        // the Bicycle class has two fields
        public int gear;
        public int speed;
```

```
// the Bicycle class has one constructor
public Bicycle(int gear, int speed)
{
    this.gear = gear;
    this.speed = speed;
}
```

```
// the Bicycle class has three methods
public void applyBrake(int decrement)
{
    speed -= decrement;
}
```

```
public void speedUp(int increment)
{
    speed += increment;
}
```

```
// toString() method to print info of Bicycle
public String toString()
{
    return ("No of gears are " + gear + "\n"
    + "speed of bicycle is " + speed);
}
```

```
// derived class
class MountainBike extends Bicycle {
```

```
// the MountainBike subclass adds one more field
public int seatHeight;
```

```
// the MountainBike subclass has one constructor
public MountainBike(int gear, int speed,
int startHeight)
{
    // invoking base-class(Bicycle) constructor
    super(gear, speed);
    seatHeight = startHeight;
}
```

```

// the MountainBike subclass adds one more method
public void setHeight(int newValue)
{
    seatHeight = newValue;
}

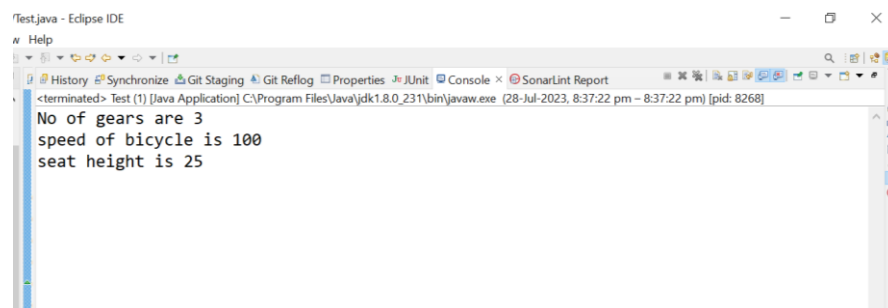
// overriding toString() method
// of Bicycle to print more info
@Override public String toString()
{
    return (super.toString() + "\n seat height is "
    + seatHeight);
}
}

// driver class
public class Test {
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}

```

### Example:



The screenshot shows the Eclipse IDE interface. The 'Console' tab is active, displaying the output of a Java application. The output text is: 'No of gears are 3', 'speed of bicycle is 100', and 'seat height is 25'. The IDE title bar indicates the file is 'Test.java - Eclipse IDE'.

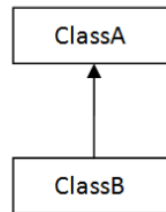
```

Test.java - Eclipse IDE
w Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (28-Jul-2023, 8:37:22 pm) [pid: 8268]
No of gears are 3
speed of bicycle is 100
seat height is 25

```

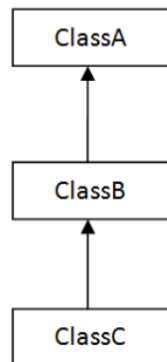
**Java supports the following types of inheritance:**

**Single Inheritance:** In single inheritance, a class can inherit from only one superclass (parent class). Java allows a class to extend a single class, forming a single parent-child relationship.



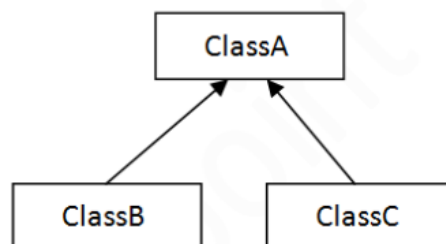
1) Single

**Multilevel Inheritance:** In multilevel inheritance, a class extends another class, which, in turn, extends another class. It forms a chain of inheritance, allowing subclasses to inherit properties and behaviors from multiple levels of ancestors.



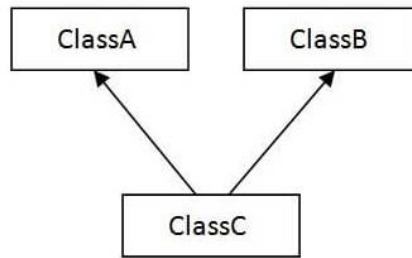
2) Multilevel

**Hierarchical Inheritance:** Hierarchical inheritance involves one superclass (parent class) being inherited by multiple subclasses (child classes). It represents a one-to-many relationship, where multiple classes share a common base class.



3) Hierarchical

**Multiple Inheritance (through Interfaces):** Multiple Inheritance is the process in which a class inherits properties from more than one class.

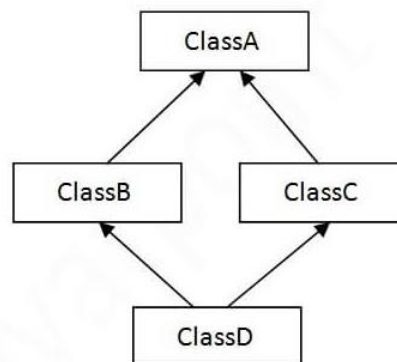


4) Multiple

Java does not support multiple inheritance to avoid the "diamond problem.". However, it provides an alternative through interfaces to achieve the same purpose. A class can implement multiple interfaces, inheriting method signatures and constants from each of them.

We will learn about interfaces in detail in upcoming sessions.

**Hybrid Inheritance:** Hybrid inheritance is a combination of two or more types of inheritance. It typically involves the use of interfaces to achieve multiple inheritance of behavior and class inheritance for single or multilevel inheritance.



5) Hybrid

It's important to note that while Java supports single inheritance for classes (extending only one class), it enables multiple inheritance of behavior through interfaces (we will cover interfaces in the upcoming sessions). This allows Java to achieve a balance between providing the flexibility of multiple inheritance and avoiding the complexities associated with multiple inheritance of state.

**Note:** We will cover interfaces in the upcoming sessions.



After 1.5 hrs of ILT, the trainer needs to assign the below lab to the students in VPL.

**Lab (30mins): (Attempt any 2)**

1. Write a program to check whether a number is a Strong number or not. Strong number is a special number whose sum of factorial of digits is equal to the original number.

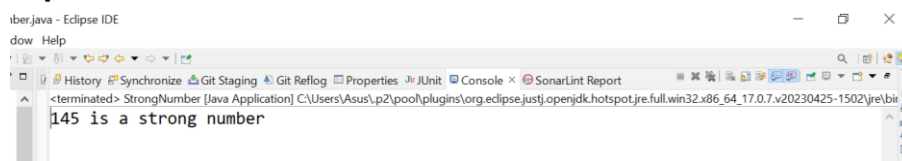
For example: 145 is a strong number. Since,  $1! + 4! + 5! = 145$

**[Hint: conditional operator, method, use parameterized method to take input]**

**Sample Input 1:**

**145**

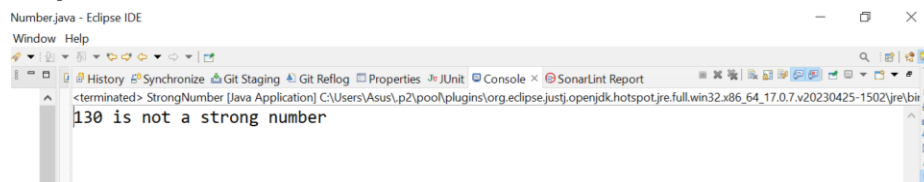
**Expected Output:**

A screenshot of the Eclipse IDE's console window. The title bar reads 'Number.java - Eclipse IDE'. The console output shows the text '145 is a strong number'.

**Sample Input 1:**

**130**

**Expected Output:**

A screenshot of the Eclipse IDE's console window. The title bar reads 'Number.java - Eclipse IDE'. The console output shows the text '130 is not a strong number'.

2. Write a program to check leap year using if else. How to check whether a given year is a leap year or not.

**[Hint: Take an input of any number. Store it in some variable say *year*.**

If a year is exactly divisible by 4 and not divisible by 100, then it is a leap year. Or if a year is exactly divisible by 400 then it is a leap year.]

**Sample Input 1:**

**2004**

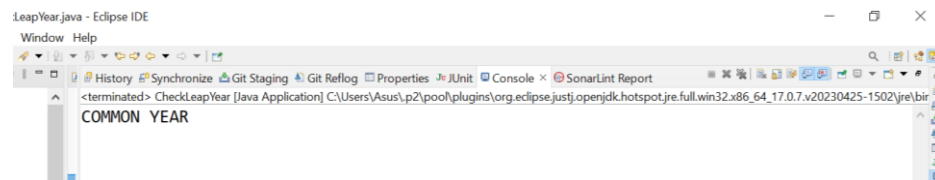
## Expected Output:



## Sample Input 2:

2023

## Expected Output:



3. Create a program to calculate the annual salary of an employee by using inheritance.

Create a class named "**User**" with the following properties and methods:

Properties:

id(int): representing the id of the User

name(String): representing the name of the User

Constructor:

Declare parameterized constructor to initialize id and name.

Create a subclass named "**Employee**" that inherits from the "User" class. Add the following additional properties and methods:

Properties:

salary(double): representing the monthly salary of the employee

Method:

double calculateAnnualSalary(): to calculate the annual salary earned by the employee.

In the main method, create an object of "Employee" class. Calculate the annual salary of the employee and display it..

[Hint:Use constructor or setter methods to set the value]

### Sample Input:

**Id:1**

**name: john**

**salary: 20000**

### Expected output:

