Types of statements in Java

eazy bytes

In Java programming, a statement denotes an operation, such as assigning the sum of variables a and b to c, outputting a message to the standard output, iterating through a list of values, conditionally executing a code block or writing data to a file etc. These statements are constructed using keywords, operators, and expressions. Depending on the action performed, statements in Java are broadly classified into three categories.

Declaration Statements

A declaration statement is used to introduce a new variable by specifying its data type and name.

```
int count = 9;
double average;
String message;
```

Expression Statements

Expressions statements comprises literals, variables, operators, and method invocations. When an expression is evaluated, it can result in the creation of a variable, the derivation of a value, or no tangible outcome.

```
9 + 6
count++
System.out.println("Hello")
```

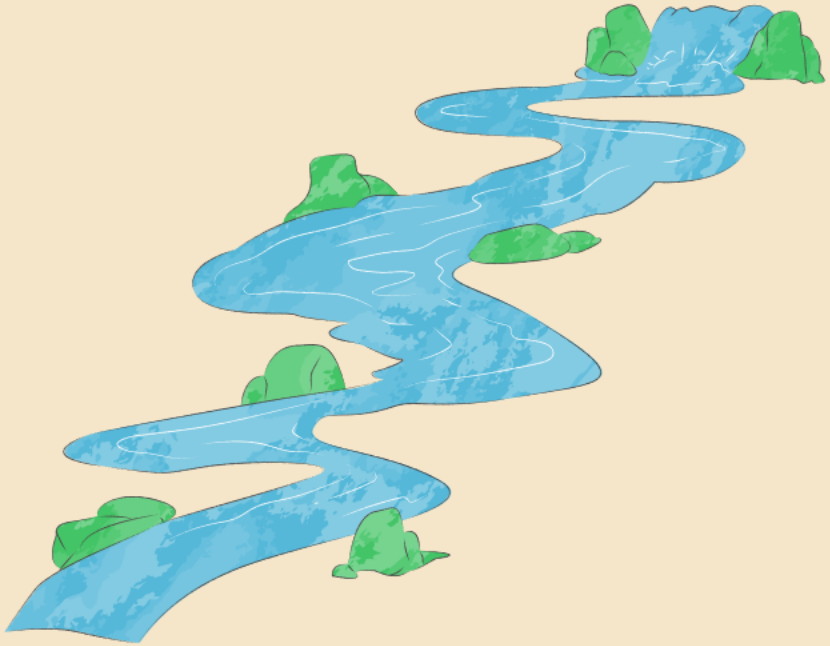
Control flow Statements

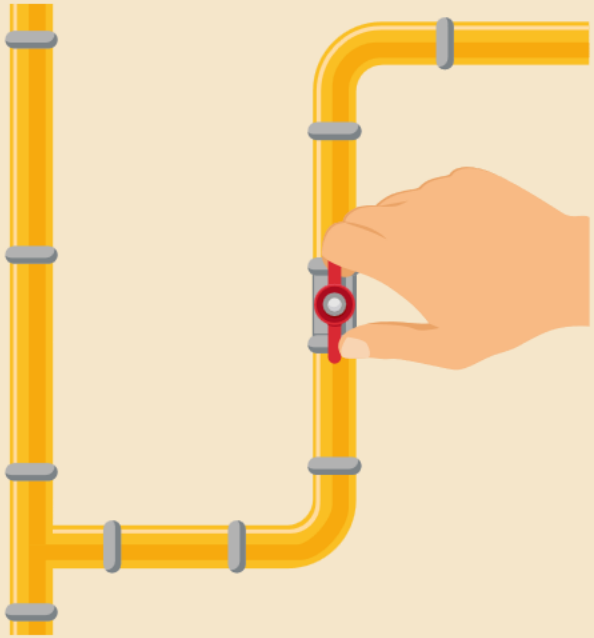
Control flow statements in Java enable actions based on conditions. They allow you to execute statements when a specific condition is met or repeat a set of statements for a specified number of times or as long as a particular condition holds true. These statements provide flexibility in managing the flow of your Java program based on varying conditions.

Control flow statements

eazy bytes

What is the difference you see in the below images ?



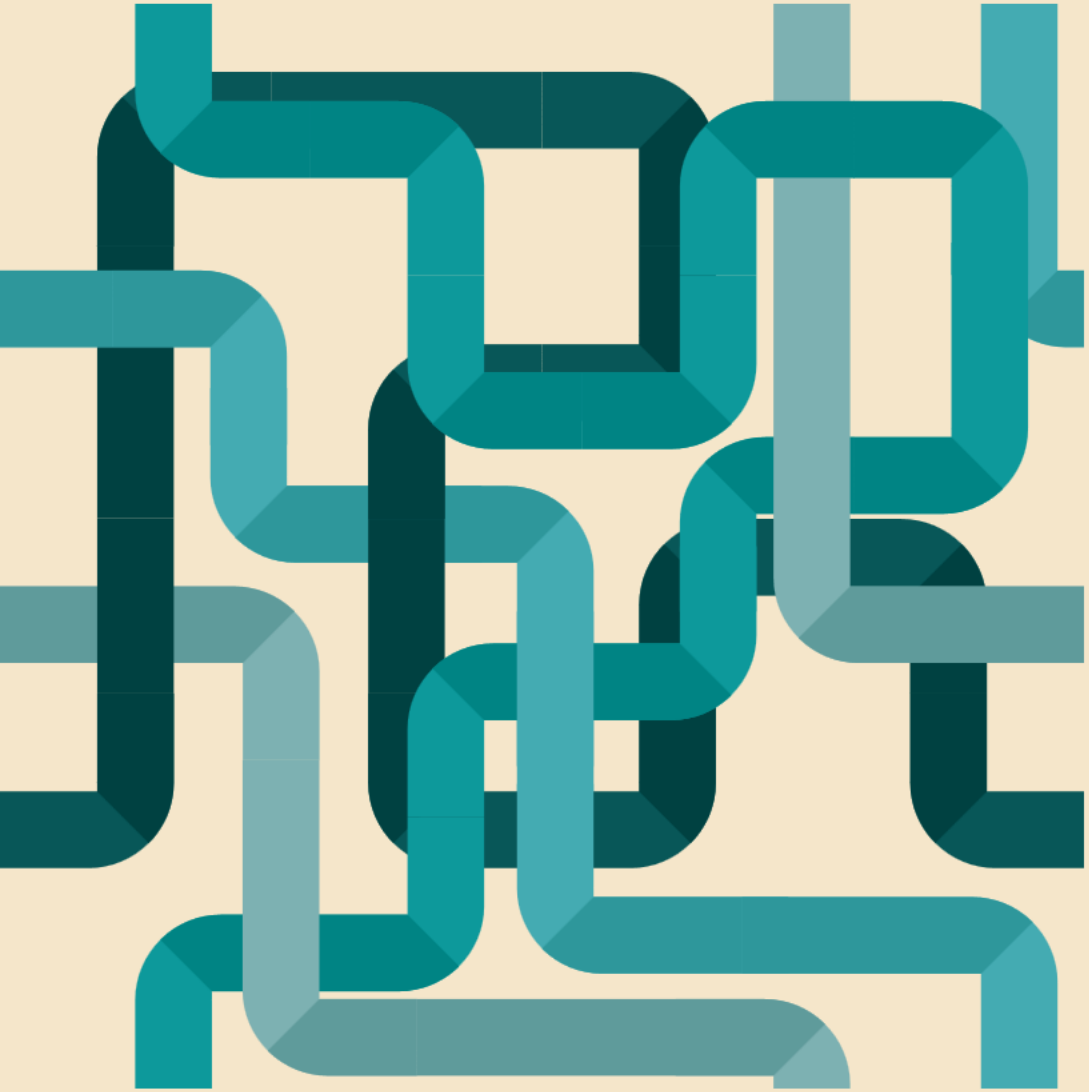


One flow is uncontrolled while the other is controlled. At times, it is necessary to manage the sequence of Java statements.

> During the execution of a Java program, each statement is executed one by one. However, certain statements must be executed conditionally, depending on the outcome of an expression evaluation. These particular statements are known as control flow statements, as computer science uses the term "control flow" (or "flow of control") to refer to the sequence in which individual statements are executed or evaluated.

Control flow statements

eazy bytes



In Java, we have the following control flow statements

- A selection statement using if-else or switch-case
- An iteration statement using for, while, or do-while
- A branching statement using break, continue, or return
- An exception-handling statement using throw, try-catch, or try-catch-finally

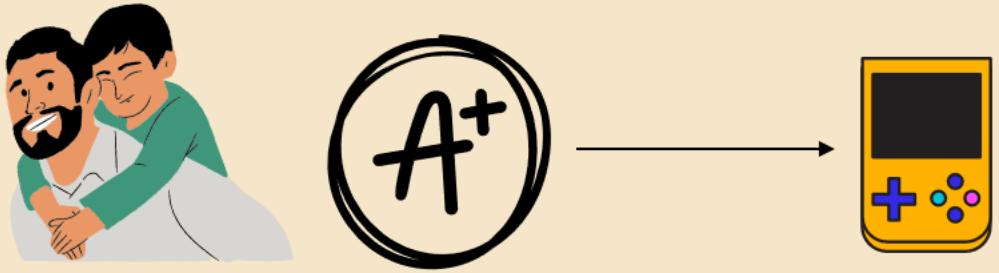
if statement

eazy bytes

In Java, the "if" statement is a control flow statement used to execute a block of code conditionally. The syntax of the "if" statement is as follows:

```
if (condition) {
    // code to be executed if condition is true
}
```

Here, the "condition" is a Boolean expression that evaluates to either true or false. If the condition is true, the code inside the block will be executed. If the condition is false, the code will be skipped.



If we want to provide an example of an if statement, we can look back to our childhood. For instance, our parents may have promised to purchase a video game console for us if we achieved an A+ grade in our final exams.

```
String myGrade = "A+";

if(myGrade.equals("A+")){
    System.out.println("Hurray, I got a Gaming console.");
}
```

Although it is feasible to omit the curly braces of an if statement when there is only a single Java statement within it, it is not advisable since it can lead to decreased code readability.

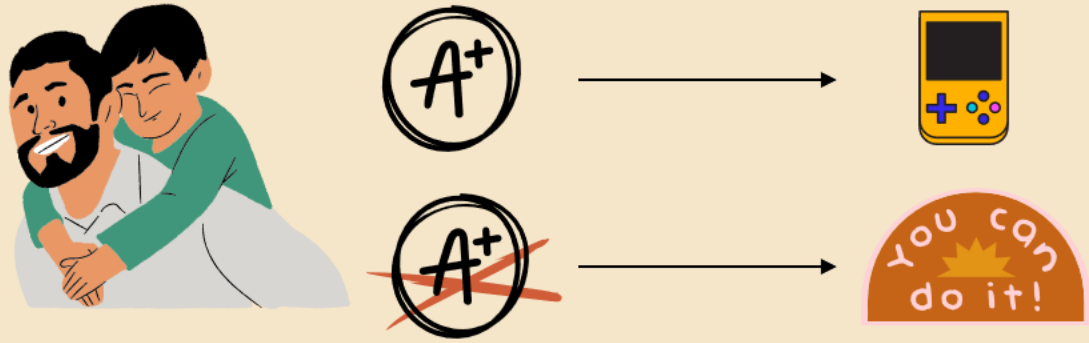
if-else statement

eazy bytes

In Java, an if-else statement is a conditional statement that allows you to execute different blocks of code based on whether a boolean expression is true or false.

```
if (condition) {
    // code to be executed if condition is true
} else {
    // code to be executed if condition is false
}
```

The code inside the first set of curly braces is executed, if the expression inside the if statement evaluates to true, otherwise the code inside the else block is executed.



Suppose we use the same example of exam grades. Our parents may have made a pledge to reward us with a video game console if we attained an A+ grade in our final exams, and if we did not, they would deliver a motivational speech instead.

```
String myGrade = "A";

if(myGrade.equals("A+")){
    System.out.println("Hurray, I got a Gaming console.");
}else {
    System.out.println("Motivation speech received");
}
```

Although it is feasible to omit the curly braces of an if & else statement when there is only a single Java statement within it, it is not advisable since it can lead to decreased code readability.

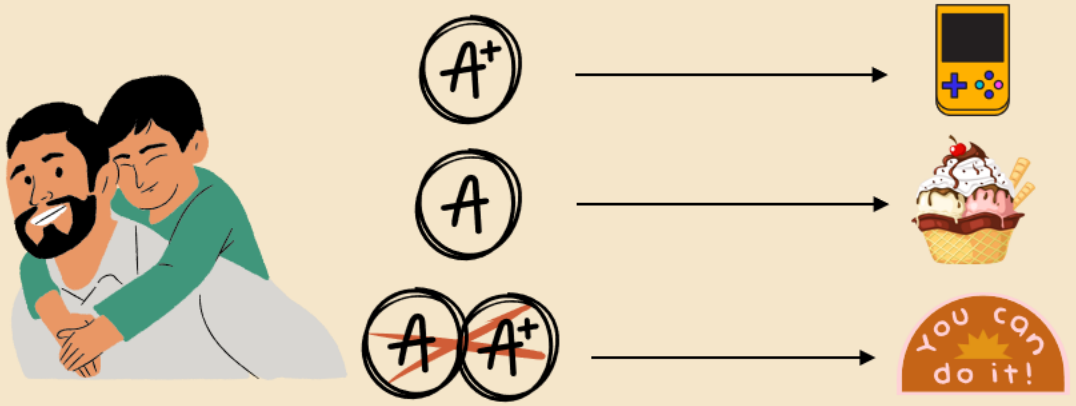
if- else if- else statement

eazy bytes

The "else if" statement is used in conjunction with the "if" statement to check for multiple conditions. The syntax for the "else if" statement is as follows:

```
if (condition1) {
    // code to be executed if condition1 is true
} else if (condition2) {
    // code to be executed if condition2 is true
} else {
    // code to be executed if all conditions are false
}
```

In this example, if the condition1 is true, the code inside the first block will be executed and the "else if" statement will be skipped. If the condition1 is false, the "else if" statement will be evaluated, and if the condition2 is true, the code inside the second block will be executed. If the condition2 is false, the code inside the "else" block will be executed.



Let's say we take the same example of exam grades. Assuming our parents made a commitment to incentivize us with a video game console if we achieved an A+ grade, or ice cream if we achieved an A grade in our annual exams, and if we didn't meet these requirements, they would give us a pep talk instead.

```
String myGrade = "A+";

if(myGrade.equals("A+")){
    System.out.println("Hurray, I got a Gaming console.");
}else if (myGrade.equals("A")) {
    System.out.println("Yummy ice cream !!!!");
} else {
    System.out.println(" Motivation speech received");
}
```

Although it is feasible to omit the curly braces of an if, else if, else statement when there is only a single Java statement within it, it is not advisable since it can lead to decreased code readability.

Nested if- else if - else statements

eazy bytes

In Java, nested if-else if - else statements involve placing one if- else if - else statement inside another. This allows for more complex decision-making based on multiple conditions. Below is an nested if-else example,

```
String dayOfWeek = "Wednesday";

if (dayOfWeek.equals("Saturday") || dayOfWeek.equals("Sunday")) {
    System.out.println("Hooray, it's the weekend!");

  if (dayOfWeek.equals("Saturday")) {
      System.out.println("Time for a relaxing day or maybe some outdoor
                          activities!");
    } else {
        System.out.println("Lazy Sunday vibes perfect for a cozy day
                            indoors.");
    }
} else {
    System.out.println("It's a weekday. Time to work or attend classes.");

    if (dayOfWeek.equals("Wednesday") || dayOfWeek.equals("Thursday")
            || dayOfWeek.equals("Friday")) {
        System.out.println("Midweek hustle! Keep going, the weekend is
                            approaching.");
    } else {
        System.out.println("Monday blues? Grab some coffee and power
                            through the day!");
    }
}
```

Ternary operator in the place of if- else statement

eazy bytes

The ternary operator in Java is a concise way to express conditional statements. It's often used as a shorthand for simple if-else statements, providing a more compact syntax. Here's an example comparing the traditional if-else statement with its ternary operator equivalent:

Using if-else statement:

```java
int x = 10;
int y;

if (x > 5) {
    y = 20;
} else {
    y = 30;
}
```

Using ternary operator:

```java
int x = 10;
int y = (x > 5) ? 20 : 30;
```

- ✓ The distinction between using the if-else statement and the ternary operator lies in their compactness. While the ternary operator allows for concise code, it's essential to note that it's not a universal replacement for all if-else statements.
- ✓ The ternary operator is suitable when both the if and else branches contain only one statement, and these statements return the same type of values. This is because the ternary operator is an expression, and it's most effective when used within expressions.
- ✓ Another distinction between using a ternary operator and an if-else statement is the ability to employ an expression containing a ternary operator as an argument to a method. In contrast, an if-else statement cannot be used directly as an argument to a method. The ternary operator, being an expression, offers flexibility in method calls, allowing for more streamlined and concise code in certain situations.

To summarize, the ternary operator is particularly useful for short and simple conditional assignments, but it may become less readable for more complex conditions or expressions. Use it judiciously to enhance code clarity and maintainability.

Why we need switch-case statement

eazy bytes

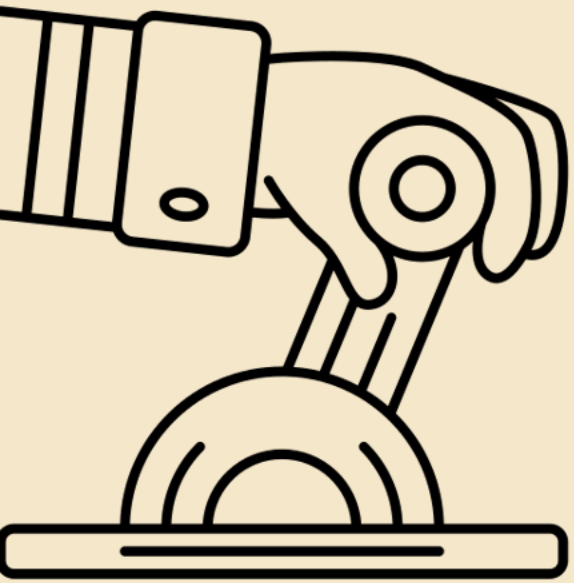
Suppose you are tasked with displaying the charges for a customer's selected fruit based on the given rates, and you are provided with the corresponding Java code that uses a series of if statements to check each possible fruit.

However, this approach can be lengthy and cumbersome. Instead, you can use a more efficient approach by checking the fruit value only once and then executing an action based on that value. Luckily, such a statement exists in Java, called a switch-case statement.

```java
String fruitName = "Mango";

if (fruitName.equals("Banana")) {
    System.out.println("$ 1.0 charged");
} else if (fruitName.equals("Apple")) {
    System.out.println("$ 1.5 charged");
} else if (fruitName.equals("Grapes")) {
    System.out.println("$ 2.0 charged");
} else if (fruitName.equals("Pineapple")) {
    System.out.println("$ 2.5 charged");
} else if (fruitName.equals("Mango")) {
    System.out.println("$ 3.0 charged");
} else {
    System.out.println("Pick a valid fruit");
}
```

switch statement in Java

eazy bytes



Below is the syntax of a switch statement,

```
switch (expression) {
    case label1:
        statements
        break;
    case label2:
        statements
        break;
    case label3:
        statements
        break;
    default:
        statements
        break;
}
```

> The expression value derived in a switch statement must be of type byte, short, char, int, enum, or String. The labels (label1, label2, etc.) within the switch statement are required to be compile-time constant expressions, and their values must fall within the range of the type of the switch value.

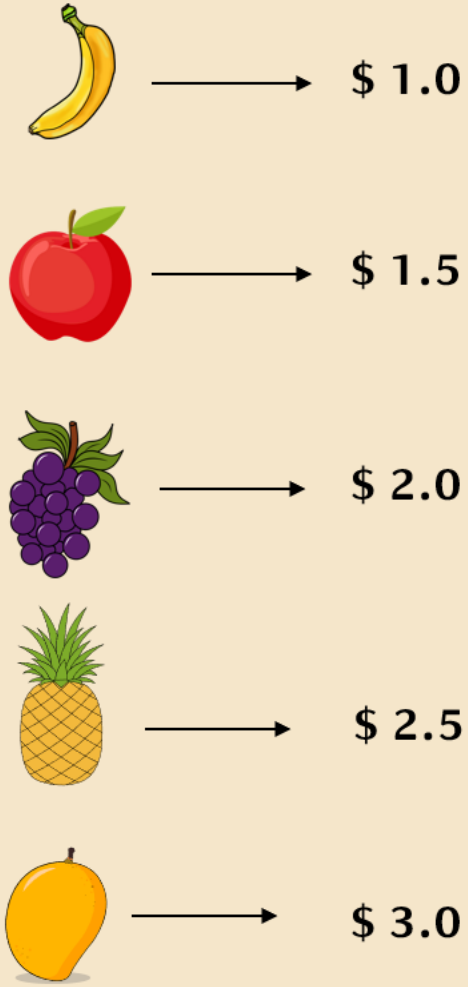
Below are the steps followed while evaluation of a switch statement:

1. The expression undergoes evaluation
2. If the value of the expression corresponds to a case label, the corresponding statements will be executed
3. If the value of the switch-value does not align with any case label, the statements under default label (if present) will be executed

switch case statement

eazy bytes

Like shown below, the switch statement is more concise and easier to read than the if-else statements.



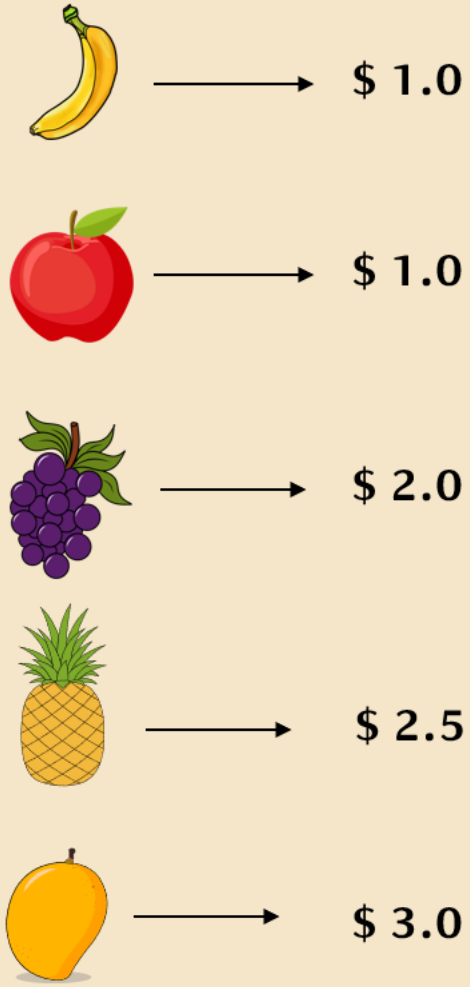

```java
String fruitName = "Mango";

switch (fruitName) {
    case "Banana":
        System.out.println("$ 1.0 charged");
        break;
    case "Apple":
        System.out.println("$ 1.5 charged");
        break;
    case "Grapes":
        System.out.println("$ 2.0 charged");
        break;
    case "Pineapple":
        System.out.println("$ 2.5 charged");
        break;
    case "Mango":
        System.out.println("$ 3.0 charged");
        break;
    default:
        System.out.println("Pick a valid fruit");
        break;
}
```

1. It is worth noting that the following primitive types cannot be used for the type of your selector variable: boolean, long, float, and double.
2. The break statement is used to exit the switch statement once the corresponding code segment is executed. Otherwise, all the following cases would be executed.
3. Technically, the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone.
4. From Java SE 7 and later only, we can use a String object in the switch statement's expression.
5. The default label does not have to be the last label to appear in a switch statement and is optional. We can define atmost one default label
6. Two case labels in a switch statement cannot be the same. It will result in compilation error.
7. Each case label in a switch statement must represent a compile-time constant. In other words, the values assigned to the labels must be ascertainable during the compilation phase. Failure to adhere to this requirement results in a compile-time error.

switch case statement

eazy bytes

The following code example, shows how a statement can have multiple case labels. For example, if you have a same price for both Banana & Apple, it is possible to combine both of them.




```java
String fruitName = "Apple";

switch (fruitName) {
    case "Banana":
    case "Apple":
        System.out.println("$ 1.0 charged");
        break;
    case "Grapes":
        System.out.println("$ 2.0 charged");
        break;
    case "Pineapple":
        System.out.println("$ 2.5 charged");
        break;
    case "Mango":
        System.out.println("$ 3.0 charged");
        break;
    default:
        System.out.println("Pick a valid fruit");
        break;
}
```

1. From Java 14, a more convenient syntax for the switch keyword is introduced with the name - switch expression.

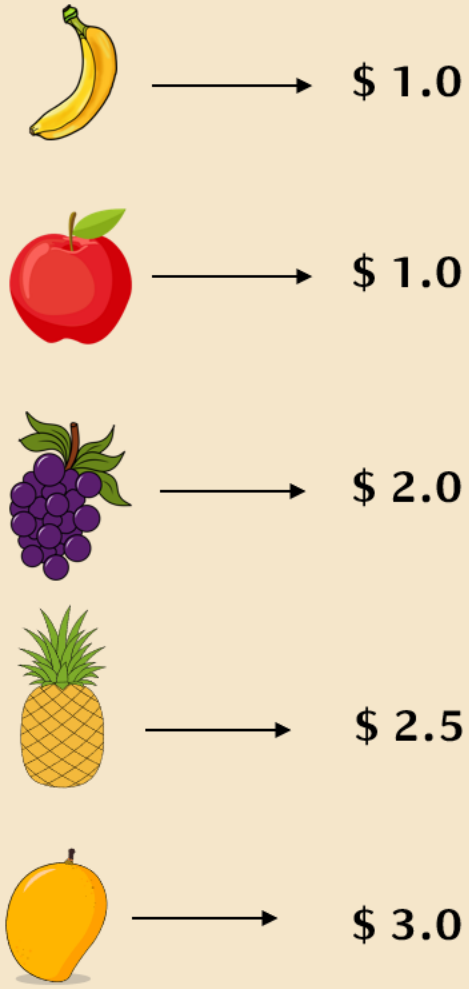
2. The syntax of switch case remains unchanged in Java 14, and its behavior remains the same. However, a new syntax for switch statements (switch expression) has been introduced in Java 14, which builds upon the existing switch case syntax.

From Java 14

switch expression

eazy bytes

Java 14 introduced a more concise form of the switch statement called switch expression. As you can see, it uses an arrow (->) and does not use a break statement, resulting in a less verbose syntax. This syntax also supports multiple constants per case, separated by commas.




```java
String fruitName = "Apple";

switch (fruitName) {
    case "Banana", "Apple" -> System.out.println("$ 1.0 charged");
    case "Grapes" -> System.out.println("$ 2.0 charged");
    case "Pineapple" -> System.out.println("$ 2.5 charged");
    case "Mango" -> System.out.println("$ 3.0 charged");
    default -> System.out.println("Pick a valid fruit");
}
```

If several lines of code have to be executed in each case, you can just put braces ({}) around the block of code.

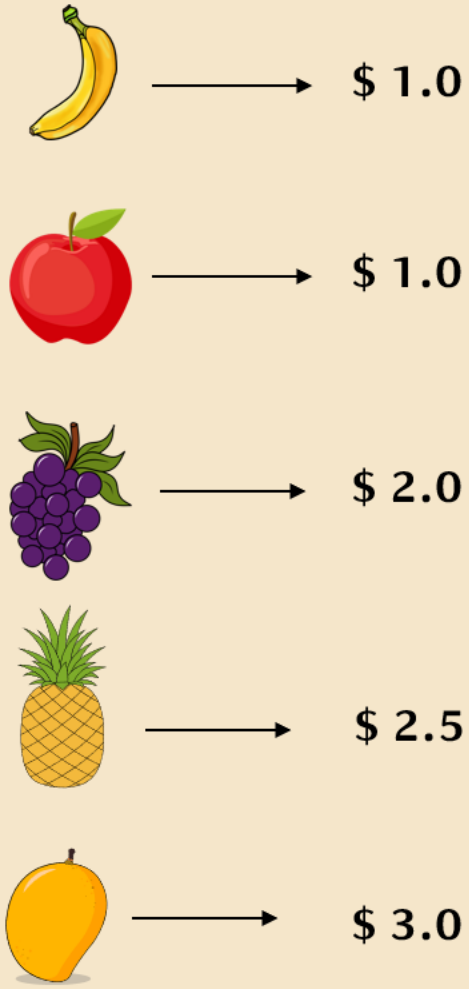
A notable advantage of utilizing switch expressions is that if there is incomplete case coverage, a compile error will be generated, ensuring comprehensive handling of all possible cases.

From Java 14

switch expression

eazy bytes

If needed, we can use switch expression to return a value like shown below,




```
String fruitName = "Apple";

String output = switch (fruitName) {
    case "Banana", "Apple" → "$ 1.0 charged";
    case "Grapes" → "$ 2.0 charged";
    case "Pineapple" → "$ 2.5 charged";
    case "Mango" → "$3.0 charged";
    default → "Pick a valid fruit";
};

System.out.println(output);
```

Using yield Keyword in switch expressions

eazy bytes

The yield keyword facilitates the termination of a switch expression by providing a value that subsequently becomes the overall value of the switch expression.

```
String day = "FRIDAY";

        int numLetters = switch (day) {

            case "MONDAY", "FRIDAY", "SUNDAY" -> {
                System.out.println(6);
                yield 6;
            }
            case "TUESDAY" -> {
                System.out.println(7);
                yield 7;
            }
            case "THURSDAY", "SATURDAY" -> {
                System.out.println(8);
                yield 8;
            }
            case "WEDNESDAY" -> {
                System.out.println(9);
                yield 9;
            }
            default -> {
                System.out.println("Invalid Day");
                yield 0;
            }

    };
```

Traditional switch statement vs switch expression

eazy bytes

Traditional switch statement

```java
String fruitName = "Apple";

switch (fruitName) {
    case "Banana":
    case "Apple":
        System.out.println("$ 1.0 charged");
        break;
    case "Grapes":
        System.out.println("$ 2.0 charged");
        break;
    case "Pineapple":
        System.out.println("$ 2.5 charged");
        break;
    case "Mango":
        System.out.println("$ 3.0 charged");
        break;
    default:
        System.out.println("Pick a valid fruit");
        break;
}
```

switch expression

```java
String fruitName = "Apple";

switch (fruitName) {
    case "Banana", "Apple" → System.out.println("$ 1.0 charged");
    case "Grapes" → System.out.println("$ 2.0 charged");
    case "Pineapple" → System.out.println("$ 2.5 charged");
    case "Mango" → System.out.println("$ 3.0 charged");
    default → System.out.println("Pick a valid fruit");
}
```

while statement

eazy bytes

In Java, a while statement is a control flow statement that allows you to repeatedly execute a block of code as long as a certain condition is true. The basic syntax of a while loop is as follows:

```
while (condition) {
  // code to be executed
}
```

The condition is a Boolean expression that is evaluated before each iteration of the loop. If the condition is true, the code inside the loop is executed. After the code inside the loop has been executed, the condition is checked again, and if it is still true, the loop repeats. This continues until the condition is false, at which point the loop terminates, and the program continues with the next statement after the while loop.

To illustrate a while statement, we can consider a common scenario from our childhood when we fall ill with the flu. In such cases, our parents would take us to the doctor who would prescribe a medicine to be taken daily until our body temperature returns to normal. This would require us to continue taking the medicine as long as our body temperature remains high, until the condition is satisfied and we are considered healthy again. This is an example of a while loop in action, where the loop continues to execute until a certain condition is met.

```
boolean isHighTemp = true;
while(isHighTemp)
{
    System.out.println("Take syrup");
    // logic to check temp & change isHighTemp
       boolean to false
}
```

while statement

eazy bytes

For example, the following code uses a while loop to print the numbers from 1 to 10:

```
int i = 1;

while (i ⩽ 10) {
    System.out.println(i);
    i++;
}
```

In this example, the while loop continues to execute as long as the variable i is less than or equal to 10. Inside the loop, the current value of i is printed, and then i is incremented by 1. The loop continues until i is equal to 11, at which point the condition i <= 10 becomes false, and the loop terminates.

while statement

eazy bytes

The condition expression in a while-loop statement is mandatory. To make a while statement an infinite loop, you need to use the boolean literal true as the condition expression:

```
while (true){

  System.out.println ("I can print infinitely");

}
```

do while statement

eazy bytes

In Java, the do-while loop is a control structure used to execute a block of code repeatedly until a specified condition is met. It is similar to the while loop, but the difference is that the do-while loop executes the code block at least once, regardless of whether the condition is true or false. The syntax for the do-while loop in Java is as follows:

```
do {
    // code block to be executed
} while (condition);
```

In this syntax, the code block is executed first, and then the condition is checked. If the condition is true, the code block is executed again, and the process is repeated until the condition is false.

Here's an example of a do-while loop which prints 6 alone as output

```
int n = 6;
do {
    System.out.println(n);//prints 6
    n++;
} while(n < 6);
```

This is because the loop body is executed at least once before the condition is checked. Since the initial value of n is 6, and 6 is not less than 5, the condition is false and the loop terminates after the first iteration. The System.out.print(n) statement within the loop body will only execute once, printing 6 to the console.

for loop

eazy bytes

In Java, the for loop is a control structure used to execute a block of code repeatedly for a fixed number of times. It is one of the most commonly used loops in Java programming and is used when the number of iterations is known in advance. The syntax for the for loop in Java is as follows:

```
for (initialization; condition; update) {
// code block to be executed
}
```

In this syntax, the initialization statement is executed before the loop starts and is used to initialize the loop variable. The condition statement is evaluated at the beginning of each iteration and is used to check whether the loop should continue. The update statement is executed at the end of each iteration and is used to update the loop variable.

We can use the childhood scenario of falling ill with the flu and visiting a doctor who prescribes medication to be taken for five days to illustrate the concept of a for loop. In this scenario, the doctor's prescription provides a clear and fixed number of times the medication should be taken. You can use a for loop to model this scenario in Java,

```
for (int day = 1; day <= 5; day++) {
    System.out.println("Taken medicine on day " + day);
}
```

In this code, the for loop iterates over the variable day, which is initialized to 1. The loop continues as long as day is less than or equal to 5, and the loop variable is incremented by 1 at the end of each iteration using the day++ statement. Within the loop body, the code System.out.println("Take medicine on day " + day) is executed.

for loop

eazy bytes

Here's an example Java code that uses a for loop to display the 5 table:

```java
for (int i = 1; i <= 10; i++) {
    int result = i * 5;
    System.out.println("5 * " + i + " = " + result);
}
```

Output

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

In this code, the loop variable i is initialized to 1, and the loop continues as long as i is less than or equal to 10. The loop variable is incremented by 1 at the end of each iteration using the i++ statement.

Within the loop body, the code int result = i * 5; multiplies the loop variable i by 5 to calculate the value of the 5 table for the current iteration. The code then uses System.out.println() to print a message to the console displaying the multiplication and the result.

Java also have a more compact form of a for statement also known as "for each" statement & "enhanced for loop". It supports a simplified way to iterate over elements in an array or any object that implements the Iterable interface. More details coming soon...

for loop

eazy bytes

In a for-loop statement, each of the three components (initialization, condition, and update expression) is optional. It's essential to highlight that the fourth component, which is the statement, is mandatory. Consequently, if there is no specific statement to execute within the for-loop, it becomes necessary to use either an empty block statement or a semicolon in place of a statement. A semicolon functioning as a statement is referred to as an empty statement or null statement.

Below are examples on how to write infinite for loops in Java,

```
for( ; ; ) {
    // An infinite for loop
}
```

```
// An infinite loop with a semicolon as a statement
for( ; ; );
```

for loop

eazy bytes

In the initialization section of a for-loop statement, you can include either a variable declaration statement, declaring one or more variables of the same type, or a list of expression statements separated by commas. It's important to note that statements within the initialization part do not conclude with a semicolon.

Furthermore, the initialization part of a for loop is executed only once when the for loop is initiated.

```
// Declaration of two variables i and j of the same type int
for(int i = 6, j = 9; ; );

// Declaration of two variables of different types
for(int i = 9, double j = 3.14; ; ); /* Compilation fails */

// Using an expression i--
int i = 10;
for(i--; ; );

// Using an expression to print a message on the console
for(System.out.println("Hello World"); ; );

// Uses two expressions: to print a message and to increment num
int i = 10;
for(i--,System.out.println("Hello World"); ; );
```

for loop

eazy bytes

// Below code compilation will be successful as we are trying to declare the i variable

```
for (int i = 9; ; );
```

// Below code compilation fails as we are trying to re-declare the i variable

```
int i = 9;
for (int i = 10; ; );
```

// Below code compilation will be successful as we are trying to reinitialize the i variable

```
int i = 9;
i = 6;
for (i = 10; ; );
```

for loop

eazy bytes

The condition expression in a for-loop statement must yield a boolean value, either true or false, or else a compile-time error will be triggered. While the condition expression is optional, its absence implies a default condition of true, leading to an infinite loop.

The expression list/update section within a for-loop statement is not mandatory. It has the flexibility to incorporate one or more expressions, each separated by a comma. However, it's important to note that only expressions capable of being converted to a statement are permissible in this context.

// Below code compilation fails as we have not written an valid expression under update section

```
for (int i = 6; i < 9 ; int j = 10);
```

// Below code compilation will be successful as i++ is a valid expression

```
for (int i = 6; i < 9 ; i++);
```

// Below code compilation will be successful as we have written valid expressions under update section

```
for (int i = 6; i < 9 ; i++, System.out.println("Hello World"));
```

Nested for loop

eazy bytes

We can also write nested for loops which means a for loop inside another for loop. This nesting can go to any level of nesting. Below are examples of the same.

This code will print the multiplication table for numbers 1 to 10. It is a good example of a nested for loop because it is clear and easy to understand. The outer loop iterates over the rows of the multiplication table, and the inner loop iterates over the columns of the multiplication table.

```java
public class MultiplicationTable {

    public static void main(String[] args) {
        int number = 10;

        for (int i = 1; i ⩽ number; i++) {
            for (int j = 1; j ⩽ number; j++) {
                System.out.println(i + " x " + j + " = " + i * j);
            }
        }
    }
}
```

Nested for loop

eazy bytes

This code will print a pyramid of asterisks. It is a good example of a nested for loop because it demonstrates how nested loops can be used to create complex shapes. The outer loop iterates over the rows of the pyramid, and the inner loop iterates over the asterisks in each row.

```java
public class Pyramid {

    public static void main(String[] args) {
        int rows = 5;

        for (int i = 1; i ⩽ rows; i++) {
            for (int j = 1; j ⩽ i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

eazy bytes

Branching statements helps to jump the flow of execution from one part of a program to another. These statements are mostly used inside the control statements and allow us to exit from a control statement when a certain condition meet.

A branching statement can be one of the following:

1. break
2. continue
3. return

break statement

eazy bytes

We already saw how break was used in switch-case statements. Similarly "break" statement is used to exit a loop prematurely based on a condition.

The labeled and unlabeled break statement are the two forms of break statement in Java.

The unlabelled "break" statement is employed to halt the execution of an inner loop or to terminate a "switch" statement. It can be utilized to terminate all loops present in Java.

A labeled "break" statement is a variant of the regular "break" statement in Java. When dealing with nested loops, if the "break" statement is used within the innermost loop, only that loop will be terminated, and not the outer loops. However, by using a labeled "break" statement, it is possible to terminate the outermost loop.

Example 1

```java
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // exit loop when i is 5
    }
    System.out.println(i);
}
```

In this example, the loop prints out the values of "i" from 0 to 4, and then exits the loop when "i" becomes 5.

In this example, the outer loop is labeled as "outerForLoop" and the inner loop is labeled as "innerForLoop". The "break" statement is used to terminate the outer loop when the value of "i" becomes 3. Since the "break" statement is labeled with "outerForLoop", it terminates the outer loop instead of just the inner loop.

Example 2

```java
// Labeling the outer loop as outerForLoop
outerForLoop:
for (int i = 1; i < 5; i++) {
    // Labeling the inner loop as innerForLoop
    innerForLoop:
    for (int j = 1; j < 3; j++) {
        System.out.println("i = " + i + " and j= " + j);
        // Terminating the outer loop
        if (i == 3) {
            break outerForLoop;
        }
    }
}
```

continue statement

eazy bytes

A continue statement can be used inside the for-loop, while-loop, and do-while statements.

The labeled and unlabeled continue statement are the two forms of continue statements in Java.

The unlabelled ”continue" statement can be used inside loops to skip the current iteration of the loop and move on to the next iteration

The labeled “continue” statement is used to control the flow of nested loops by skipping the rest of the current iteration of an outer loop and proceeding to the next iteration. It is particularly useful when working with nested loops, allowing you to specify which loop to continue.

Example 1

```java
for (int i = 0; i < 10; i++) {

    if (i % 2 == 0) {
        continue; // skip even numbers
    }
    System.out.println(i);

}
```

In this example, the loop prints out all of the odd numbers between 0 and 9. If the current value of "i" is even, the "continue" statement is executed, which skips the current iteration of the loop and moves on to the next iteration.

In this example, the labeled continue statement continue outer; is used to skip the rest of the outer loop. This means that when the condition j == 2 is met, the loop will immediately jump back to the beginning of the outer loop and continue iterating.

Example 2

```java
outer:
for (int i = 0; i < 5; i++) {
    inner:
    for (int j = 0; j < 5; j++) {
        if (j == 2) {
            continue outer; // This will skip the rest of the outer loop
        }
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

return statement

eazy bytes

return

The "return" statement is used to exit a method and return a value. Here's an example:

```java
public int add(int x, int y) {
    int sum = x + y;
    return sum;
}
```

In this example, the "add" method takes two integers as parameters and returns their sum. The "return" statement is used to exit the method and return the value of "sum".

return

Using "return" statement inside a if block, for loop or while loops will complete the execution, return the values & exit the method.

```java
public static String getStudentGrade(int marks) {
    if (marks ⩾ 40) {
        return "Passed";
    } else {
        return "Failed";
    }
}
```

In this example, there are two return statements inside the method, but still compiler will not complain as at any case only one of them will be executed.

Local variables & Scope

eazy bytes

In Java, a local variable is a variable that is declared inside a method or a block, and its scope is limited to the block or method in which it is declared. Local variables cannot be accessed outside of their scope.

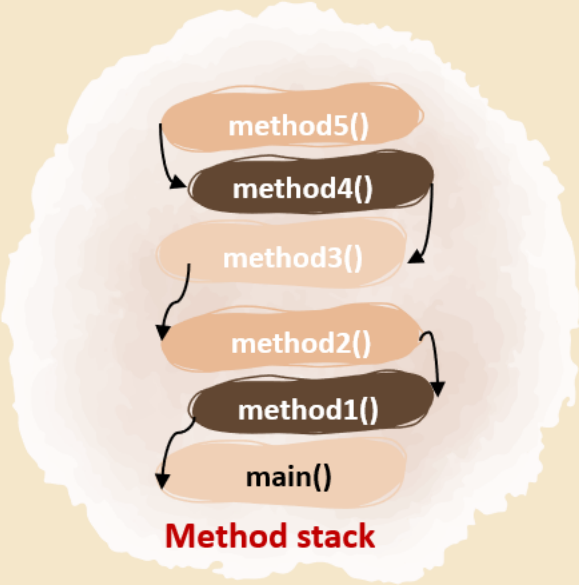
The scope of a local variable begins at the point where it is declared and ends at the end of the block in which it is declared. For example:

```
public void myMethod() {
  int x = 10; // x is a local variable
  System.out.println(x);
} // end of method, x no longer exists
```

In this example, x is a local variable that is declared inside the myMethod() method. Its scope is limited to the method, and it can only be accessed within the method. Once the method ends, x is no longer accessible.

It is important to note that local variables can have the same name as instance variables or class variables, but they are completely separate and unrelated variables with their own scope.

Local variables are stored on the stack, which is a special region of memory used for storing local variables and method parameters. The stack is a last-in, first-out (LIFO) data structure, and when a method is called, a new frame is created on the stack to hold the local variables and parameters for that method. When the method returns, its frame is removed from the stack, and its local variables are destroyed.



Local variables & Scope

eazy bytes

Here's an example of local variables and scope using a for loop in Java:

```
public void myMethod() {
    for (int i = 0; i < 5; i++) {  // i decalred as a local variable
        int x = i * 2; // x is a local variable
        System.out.println("i = " + i + ", x = " + x);
    }
    // x & i are not accessible outside for loop
}
```

Here's an example of an if-else block in Java with a local variable:

```
int number = 10;

if (number > 0) {
    int square = number * number;
    System.out.println("The square of " + number + " is " + square);
} else {
    int absoluteValue = -1 * number;
    System.out.println("The absolute value of " + number + " is " + absoluteValue);
}

// Compilation error: Cannot find symbol 'square'
System.out.println("The square of the number is " + square);
```

Local variables & Scope inside switch block

eazy bytes

In Java, local variables can be declared inside a switch block. However, the scope of a local variable declared inside a switch block can be a bit tricky to understand, as it depends on the location of the variable declaration within the switch block.
Here's an example of a switch block in Java with a local variable:

```
int number = 2;

switch (number) {
    case 1:
        System.out.println("Odd number");
        System.out.println(i); // Compilation error as i is never declared
        break;
    case 2:
        int i = number; // local variable i declared first time
        System.out.println(i);
        System.out.println("Even number");
        break;
    default:
        i = 10; // local variable i can be accessed as it is declared in the top
        System.out.println(i);
        System.out.println("Invalid number");
        break;
}

System.out.println(i); // Compilation error as i is out of the scope
```

Local variables & Scope

eazy bytes

When declaring local variables in Java, it is important to follow best practices to ensure code readability and maintainability. Here are some best practices to follow:

Declare variables with the narrowest scope possible: Only declare variables within the block of code where they are needed. This helps to prevent naming conflicts and reduces the risk of accidental modification of the variable value.

Use meaningful variable names: Choose variable names that accurately describe the data they represent. This helps to make the code more understandable and reduces the likelihood of errors.

Initialize variables when they are declared: Always initialize local variables when they are declared.
This helps to prevent the occurrence of undefined behavior and unexpected errors due to uninitialized variables.

Declare one variable per line: Declare only one variable per line to make the code more readable and maintainable.

Avoid declaring unnecessary variables: Only declare variables that are actually needed in the code. Unnecessary variables can add complexity and confusion to the code.

Keep variable declarations close to their first use: Declare variables as close to their first use as possible to reduce the risk of naming conflicts and make the code easier to understand.