Contents

OOPS concepts

The four main concepts of Java Object-Oriented Programming (OOP) are:

1. **Encapsulation**: This is the concept of wrapping data (variables) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of an object's components, which is why we use getters and setters. This protects the internal state of the object and helps in achieving data hiding.
2. **Inheritance**: Inheritance allows a new class (subclass) to inherit properties and behaviors (methods) from an existing class (superclass). This promotes code reusability and helps in establishing a relationship between different classes (parent-child relationship). For example, a `Dog` class can inherit from an `Animal` class.
3. **Polymorphism**: Polymorphism allows methods or functions to take on multiple forms. It comes in two types:
   - **Compile-time polymorphism** (method overloading): Multiple methods in the same class with the same name but different parameters.
   - **Runtime polymorphism** (method overriding): A subclass provides a specific implementation of a method that is already defined in its superclass.
4. **Abstraction**: Abstraction hides the complex implementation details and shows only the necessary features of an object. In Java, abstraction is achieved using abstract classes and interfaces. It allows focusing on what an object does rather than how it does it. For example, a `Vehicle` interface might have an abstract method `start()`, which is implemented differently by `Car` and `Bike` classes.

Difference between encapsulation and abstraction

Encapsulation:

- **Definition**: Encapsulation is the practice of bundling the data (attributes) and methods (functions) that operate on that data into a single unit or class. It restricts direct access to certain components of an object and protects its internal state from unintended modifications.
- **Focus**: Encapsulation is concerned with controlling access to the data within an object.
- **How it works**: It achieves data hiding by making variables private or protected and providing public getter and setter methods to manipulate or access the data safely.

Abstraction:

- **Definition**: Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. It focuses on "what" an object does rather than "how" it does it.
- **Focus**: Abstraction is concerned with hiding unnecessary details and providing a simple interface for interaction.
- **How it works**: It is achieved through abstract classes and interfaces that define methods without implementation. The concrete classes provide the specific implementation.
- **Example**: A `Vehicle` interface has an abstract method `start()`. The concrete classes `Car` and `Bike` implement this method differently.

Key Differences:

- **Encapsulation**: Focuses on **how** data is accessed and protected, using access modifiers to control visibility (e.g., private, public).
- **Abstraction**: Focuses on **what** functionality an object provides, hiding complex logic through abstract methods or interfaces.

Memory Allocation in Java

1. **Heap Memory:**
   - **Arrays:** Arrays, whether they are arrays of primitive types or arrays of objects, are allocated on the heap. The heap is a region of memory used for dynamic memory allocation, and it is managed by the Java Virtual Machine (JVM). The heap memory is shared among all threads.
   - **Objects:** All objects are also allocated on the heap. This includes instances of user-defined classes.
2. **Stack Memory:**
   - **Local Variables:** Local variables (including primitives and references to objects) declared within methods are stored on the stack. Each thread has its own stack, which is used to store frames. Each frame contains local variables, the operand stack, and the frame data.
   - **Method Call Frames:** When a method is called, a new frame is created on the stack for that method call. This frame contains all the local variables and the call's execution state.

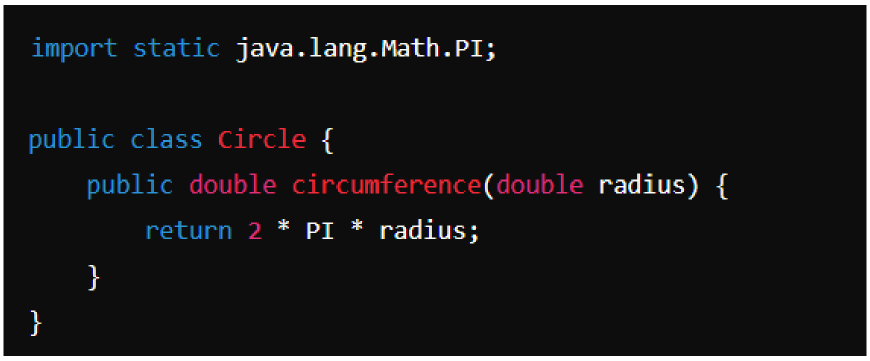
Detailed Explanation

- **Heap Memory:**
  - Heap memory is used for dynamic memory allocation where objects and arrays reside. This memory is managed by the garbage collector, which automatically reclaims memory that is no longer in use.
  - Arrays in Java are considered objects. When you create an array, the JVM allocates memory for the array on the heap and initializes it to the default values for the array's element type.
- **Stack Memory:**
  - Stack memory is used for method execution and contains method-specific data such as local variables and partial results. Each time a method is invoked, a new frame is pushed onto the stack.
  - Local variables within a method are stored in the stack frame of that method. These local variables can be primitive types or references to objects (the actual objects reside on the heap).

In Java, class variables (also known as **static variables**) are not stored in the heap. They are stored in a special area of memory called the **method area**, which is part of the **Java Virtual Machine (JVM)**. The method area holds class-level data, including static variables, method bytecodes, and other metadata related to classes.

Static Imports

In Java, static import is a feature that allows members (fields and methods) defined in a class to be used in another class without specifying the class they belong to.



```java
import static java.lang.Math.PI;

public class Circle {
    public double circumference(double radius) {
        return 2 * PI * radius;
    }
}
```

Only static members of a class can be imported using static imports . we cannot import non static members with this.

Autoboxing and Unboxing in Java

In Java, primitive data types are treated differently so do there comes the introduction of wrapper classes where two components play a role namely Autoboxing and Unboxing. Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

Passed as a parameter to a method that expects an object of the corresponding wrapper class.

Assigned to a variable of the corresponding wrapper class.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

Passed as a parameter to a method that expects a value of the corresponding primitive type.

Assigned to a variable of the corresponding primitive type.

Constructors

Default constructors

If we do not create a constructor then java compiler creates a no argument constructor for us by default.

We can create multiple constructors
if make only one default constructor and make it private then that class cannot be instantiated ….this is epful in for classes whose all methos are abstract.

This keyword

The this keyword is a reference variable in Java that points to the current object. It is primarily used within a class to differentiate between instance variables and parameters, invoke constructors, and pass the current object as an argument. Here’s a breakdown of its usage:

1. Referencing Instance Variables:

The this keyword is used to resolve naming conflicts between instance variables and parameters. When an instance variable has the same name as a parameter, this helps clarify that the instance variable is being referenced.

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

    public Person(String firstName, String lastName) {
        // Using `this` to refer to instance variables
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

2. Calling Another Constructor:

The this() keyword is used to call another constructor within the same class. This is useful for constructor chaining, where one constructor calls another constructor to reuse code.

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

    // Default constructor calls parameterized constructor
    public Person() {
        this("Anannya", "Hiteshi");
    }

    // Parameterized constructor
    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Key Points:

The this() call must be the first statement in the constructor.

It allows constructors to reuse code and reduce redundancy.

3. Passing the Current Object:

The this keyword can be used to pass the current object as an argument to methods or constructors.

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

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

    // Method that takes Person object as parameter
    public void printPerson(Person p) {
        System.out.println("First Name: " + p.firstName);
        System.out.println("Last Name: " + p.lastName);
    }

    // Passing current object using `this`
    public void display() {
        printPerson(this);
    }
}
```

4. Returning the Current Object:

The this keyword can be used to return the current instance from a method, allowing method chaining.

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

    public Person setFirstName(String firstName) {
        this.firstName = firstName;
        return this; // returning current object
    }

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

The `static` Keyword in Java

The `static` keyword in Java is used to indicate that a particular member (variable, method, or nested class) belongs to the class itself rather than to instances of the class. This means that:

1. **Static Variables:**
   - A static variable is shared among all instances of the class. There is only one copy of the static variable in memory, regardless of how many instances of the class are created.
   - Static variables can be accessed directly using the class name (e.g., `ClassName.variableName`), without needing an instance of the class.

```
2. class Example {
       static int counter = 0;

       public Example() {
           counter++;   // Incrementing the static variable
       }
   }

   class Test {
       public static void main(String[] args) {
           Example obj1 = new Example();
           Example obj2 = new Example();
           System.out.println(Example.counter); // Outputs: 2
       }
   }
```

3. **Static Methods:**
   - Static methods can be called without creating an instance of the class. They can only directly access other static members (variables and methods) of the class.

```
4. class MathUtils {
       static int square(int number) {
           return number * number;
       }
   }

   class Test {
       public static void main(String[] args) {
           int result = MathUtils.square(5);  // Calling static method
           System.out.println(result);  // Outputs: 25
       }
   }
```

5. **Static Blocks:**
   - Static blocks are used to initialize static variables. They are executed when the class is loaded into memory, before any objects of the class are created and before any static methods are called.

Static Initialization Blocks

Static initialization blocks, also known as static blocks, are used for initializing static variables or performing other setup tasks at the time of class loading. These blocks run only once, when the class is first loaded into memory.

Characteristics of Static Blocks:

- Multiple static blocks can be used in a class, and they are executed in the order they appear in the code.
- Static blocks are executed before the main method or any constructors of the class.

Example of Static Initialization Blocks:

```
public class Example {
    static int value;

    // First static block
    static {
        value = 5;
        System.out.println("First static block: value = " + value);
    }

    // Second static block
    static {
        value = 10;
        System.out.println("Second static block: value = " + value);
    }

    public static void main(String[] args) {
        System.out.println("Main method: value = " + value);
    }
}
```

Explaination:

1. **Static Blocks Execution:**
   - When the `Example` class is loaded, the static blocks are executed in the order they appear.
   - The first block sets `value` to 5 and prints the message.
   - The second block then sets `value` to 10 and prints a new message.
2. **Output:**
   - The static blocks run before the `main` method, so the value of `value` when the `main` method is executed is 10, the last value assigned by the static blocks.

Output:

```
sql
Copy code
First static block: value = 5
Second static block: value = 10
Main method: value = 10
```

Usage:

- Static blocks are useful for initializing static variables with complex logic or for performing setup tasks that need to happen when the class is first loaded.
- However, static blocks should be used carefully as they run before the class is fully initialized and can make the code harder to read if overused.

Order of Initialization

1. If there is superclass initialize it first.
2. Static variable declarations and static initializers in order they appear in the file
3. Instance variable declarations and instance initializers in order they appear in the file
4. The constructor.

Abstraction in java

Abstract Class:

An **abstract class** in Java is a class that cannot be instantiated on its own. It is used as a blueprint for other classes and may contain abstract methods (methods without a body) as well as concrete methods (with a body). Abstract classes provide partial abstraction and are typically used when classes share some common behavior but also need some specific implementation.

Key Features of Abstract Class:

1. **Partial abstraction**: It can have both abstract (without implementation) and non-abstract (with implementation) methods.
2. **Constructor and instance variables**: Abstract classes can have constructors, member variables, and methods like regular classes.
3. **Inheritance**: A class extends an abstract class using the `extends` keyword. The subclass must implement all abstract methods or be declared abstract itself.
4. **Single inheritance**: A class can inherit only one abstract class (Java doesn't support multiple inheritance with classes).

Example:

```
java
Copy code
abstract class Animal {
    // Abstract method (no body)
    public abstract void sound();

    // Concrete method
    public void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.sound(); // Outputs: Dog barks
        dog.sleep(); // Outputs: Animal is sleeping
    }
}
```

In this example, `Animal` is an abstract class, and the concrete classes `Dog` and `Cat` implement the abstract method `sound()`.

Interface:

An **interface** in Java is a completely abstract type that is used to specify a set of methods that a class must implement. It contains only abstract methods (before Java 8) and constants. Interfaces are used to achieve full abstraction and to define the behavior that a class must implement.

Key Features of Interface:

1. **Full abstraction**: Before Java 8, all methods in an interface are abstract (without implementation). Since Java 8, it can also have default methods (with a body) and static methods.
2. **No constructors or instance variables**: Interfaces cannot have constructors or instance variables; only constant variables (final and static).
3. **Multiple inheritance**: A class can implement multiple interfaces, thus achieving multiple inheritance in Java.
4. **Implementation**: A class implements an interface using the `implements` keyword and must provide implementations for all abstract methods of the interface.

Example:

```
java
Copy code
interface Vehicle {
    void start();  // abstract method
}

interface FuelVehicle {
    void refuel();
}

class Car implements Vehicle, FuelVehicle {
    public void start() {
        System.out.println("Car starts");
    }

    public void refuel() {
        System.out.println("Car is refueled");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();   // Outputs: Car starts
        myCar.refuel();  // Outputs: Car is refueled
    }
}
```

In this example, `Vehicle` and `FuelVehicle` are interfaces, and `Car` implements both of them. This demonstrates multiple inheritance, where a class can implement more than one interface.

Key Differences Between Abstract Class and Interface:

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods	Only abstract methods (Java 8 introduced default/static)
Constructors	Can have constructors	Cannot have constructors
Variables	Can have instance variables and constants	Can only have static and final variables
Multiple inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Use case	When classes share common functionality but need some specific implementations	When unrelated classes need to share a common behavior

In short, use **abstract classes** when you need to share behavior between closely related classes, and use **interfaces** when you want to specify a contract that can be applied to unrelated classes.

Arrays methods

Arrays.sort();

- The complexity of the **Arrays.sort(a)** method in Java depends on the sorting algorithm used by the implementation. The **Arrays.sort(a)** method typically uses a modified version of the Quicksort algorithm for primitive types and a modified version of the TimSort algorithm for objects.

- For primitive types (**int**, **char**, etc.), the **Arrays.sort(a)** method uses a dual-pivot Quicksort algorithm, which has an average-case time complexity of O(n log n), where n is the number of elements in the array. In the worst case, the dual-pivot Quicksort algorithm has a time complexity of O(n^2), but this is very rare and occurs only for specific input patterns.

- For objects, the **Arrays.sort(a)** method uses a TimSort algorithm, which is a hybrid sorting algorithm combining merge sort and insertion sort. The TimSort algorithm has an average-case time complexity of O(n log n) and guarantees stable sorting (i.e., the relative order of equal elements is preserved).

- In both cases, the space complexity of the **Arrays.sort(a)** method is O(log n), indicating that it requires additional memory for recursion or temporary storage during the sorting process.

- It's worth noting that the actual performance of **Arrays.sort(a)** can vary depending on factors such as the size of the array, the data distribution, and the specific implementation used by the Java runtime environment. However, in general, it provides an efficient and reliable way to sort arrays in Java.

Arrays.equals()

This method is used to compare the elements of two arrays to check if they contain the same elements

Example : Arrays.equals(arr1, arr2)

Arrays.fill()

This method is used to fill an array with a given value

Example : int arr[] = new int[4];

Arrays.fill(arr, 1)

The new array will be : [1,1,1,1]

Arrays.asList()

This method is used to return an array as a list …basically convert an array to a list

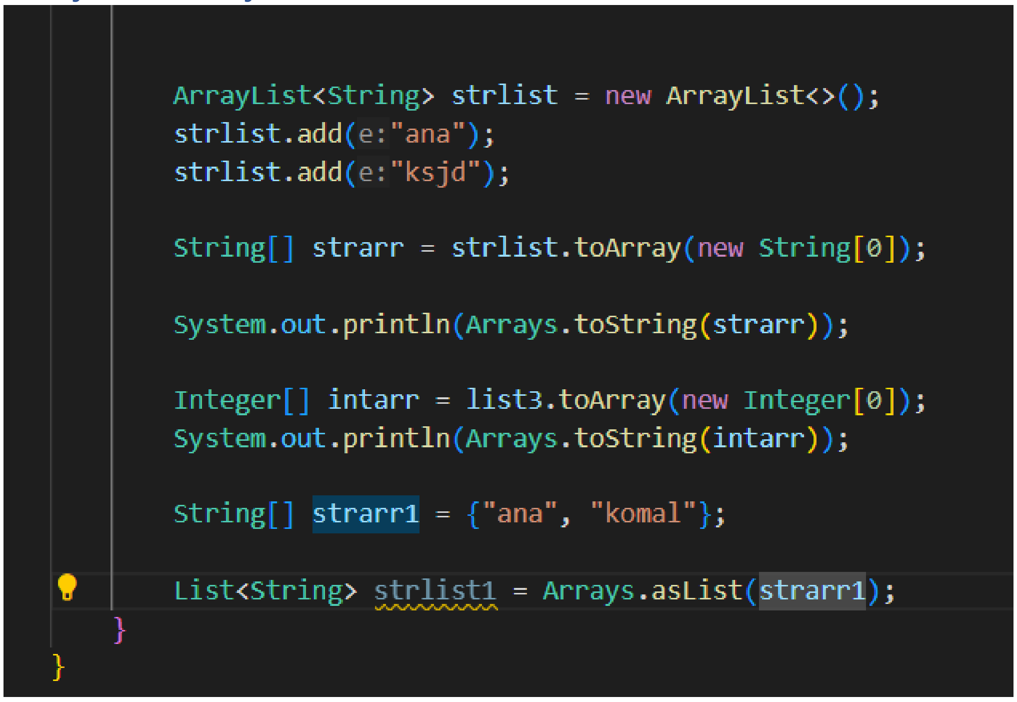
```
List<Integer> list = Arrays.asList(arr)
```

Arrays.toString()

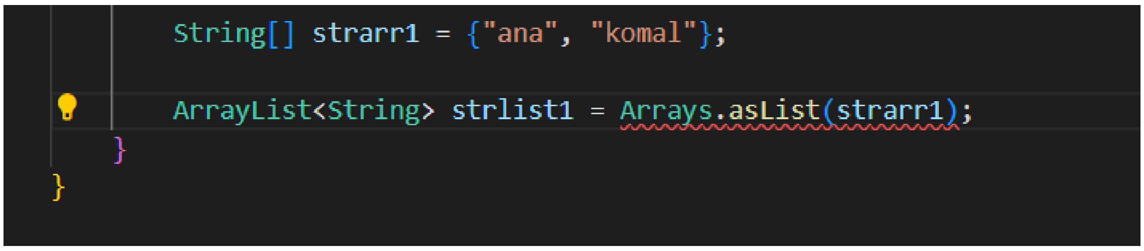
```
System.out.println(Arrays.toString(arrName)); //prints the whole array as => [element1, element2, …]
```

ArrayList

Arrays and ArrayList conversion



```
        ArrayList<String> strlist = new ArrayList<>();
        strlist.add(e:"ana");
        strlist.add(e:"ksjd");

        String[] strarr = strlist.toArray(new String[0]);

        System.out.println(Arrays.toString(strarr));

        Integer[] intarr = list3.toArray(new Integer[0]);
        System.out.println(Arrays.toString(intarr));

        String[] strarr1 = {"ana", "komal"};

        List<String> strlist1 = Arrays.asList(strarr1);
    }
}
```



```
        String[] strarr1 = {"ana", "komal"};

        ArrayList<String> strlist1 = Arrays.asList(strarr1);
    }
}
```

It looks like there's a type mismatch in your code. The `Arrays.asList` method returns a `List`, not an `ArrayList`, so you cannot directly assign it to an `ArrayList` variable.

Raw Types vs. Generics in ArrayList

Raw Types

When you use raw types, you don't specify the type of objects the collection should contain. For example:

```
java
Copy code
ArrayList list1 = new ArrayList();
list1.add("Hello");
list1.add(1); // Adding an integer
```

With raw types, the compiler cannot enforce type constraints. As a result, you can mix different types within the same collection, which can lead to issues when you later try to retrieve and cast these elements.

Example of potential runtime error:

```
java
Copy code
ArrayList list1 = new ArrayList();
list1.add("Hello");
list1.add(1);

String s = (String) list1.get(1); // This will throw ClassCastException at runtime
```

In the above example, retrieving an `Integer` from the list and casting it to a `String` will result in a `ClassCastException` at runtime.

Generics

Generics provide a way to specify the type of objects that a collection can contain, thereby enforcing type safety at compile time. For example:

```
java
Copy code
ArrayList<String> list1 = new ArrayList<>();
list1.add("Hello");
// list1.add(1); // Compile-time error: incompatible types
```

Here, `ArrayList<String>` specifies that the list can only contain `String` objects. The compiler enforces this constraint, so attempting to add an `Integer` will result in a compile-time error rather than a runtime error.

Example Comparison

Using Raw Types:

```
java
Copy code
ArrayList list1 = new ArrayList();
list1.add("Hello");
list1.add(1);

String s = (String) list1.get(1); // Runtime error: ClassCastException
```

Using Generics:

```
java
Copy code
ArrayList<String> list1 = new ArrayList<>();
list1.add("Hello");
// list1.add(1); // Compile-time error: incompatible types
```

Summary

- **Raw Types**: Allow any type of object, which can lead to runtime errors if objects are incorrectly cast or used.
- **Generics**: Enforce type safety at compile time, reducing the risk of runtime errors related to type mismatches.

By using generics, you ensure that only objects of the specified type can be added to the collection, catching type errors early during compilation and preventing potential runtime issues.

concurrent modification error

We can not iterate and modif a list at same time foreac loop gives concurrent modification error

Iterating ArrayList

Iterating Over `ArrayList`

1. For Loop

```
for (int i = 0; i < list.size(); i++) {
        Integer number = list.get(i);
        System.out.println(number);
    }
```

2. Enhanced For Loop

```
  for (Integer number : list) {
        System.out.println(number);
    }
```

3. Iterator

```
 for(Iterator<Integer> iterator = list3.iterator(); iterator.hasNext();){
            Integer number = iterator.next();
            System.out.println();
            iterator.remove();
        }
```

How `Iterator` Works:

- **Initialization**: `Iterator` is created with `list.iterator()`, initially pointing before the first element.
- **Traversal**: `iterator.next()` moves the cursor to the next element and returns it. Calling `next()` advances the cursor.
- **Removal**: `iterator.remove()` safely removes the last element returned by `next()`. This avoids `ConcurrentModificationException`, which can occur in normal loops if the list is modified directly during iteration.

Notes:

- **Iterator**: Safely handles concurrent modification when removing elements during iteration, unlike traditional loops.

When it is said that the `Iterator` "points between elements," it means:

- **Initial State**: When an `Iterator` is first created, it is positioned before the first element of the collection. It does not point to any specific element yet.
- **During Iteration**: As you call `iterator.next()`, the iterator moves to the next element. After calling `next()`, the iterator points to the element just returned and is positioned between this element and the next one.

Using left shift operator to find the power of 2;

The shift operators in Java (<<, >>, and >>>) are used for shifting the bits of a number to the left or right. In the context of calculating powers of 2, the left shift operator (<<) is commonly used.

The left shift operator (<<) shifts the bits of a number to the left by a specified number of positions. Each shift to the left effectively multiplies the number by 2. Here's how it works:

1. Start with the binary representation of a number that is a power of 2. For example, **1** is **0001**, **2** is **0010**, **4** is **0100**, and so on.
2. Applying the left shift operator (<<) to a power of 2 shifts all the bits to the left by the specified number of positions. For example, **1 << 1** shifts **0001** one position to the left, resulting in **0010**, which is the binary representation of **2**. Similarly, **1 << 2** shifts **0001** two positions to the left, resulting in **0100**, which is the binary representation of **4**.

By using the left shift operator repeatedly, you can calculate higher powers of 2. For example, **1 << 3** would give **1000**, which is the binary representation of **8**.

In the updated code snippet provided earlier, **(1 << temp)** is used to calculate the value of **2** raised to the power of **temp**. It effectively performs a left shift operation on the binary representation of **1** by **temp** positions, which results in the binary representation of the corresponding power of 2.

By using the left shift operator (<<), the code can efficiently check if a number is divisible by a power of 2 and calculate the largest power of 2 that divides the given number **x**.

Scanner

scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

The code snippet you provided seems to be written in Java.

The line **scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");** is using the **skip** method of a **Scanner** object. The regular expression **(\r\n|[\n\r\u2028\u2029\u0085])?** is passed as the argument to the **skip** method.

This regular expression is used to skip any line terminators or newlines present in the input stream being scanned. The line terminators it matches are:

- **\r\n**: Carriage return followed by a newline (Windows style)
- **[\n\r\u2028\u2029\u0085]**: Any of the characters newline, carriage return, line separator, paragraph separator

The **?** at the end of the regular expression makes the entire expression optional, meaning it will skip zero or one occurrence of the line terminators or newlines.

In summary, this line of code is used to skip over any line terminators or newlines present in the input stream being scanned by the **Scanner** object

Scanner.hasNextLine();

To read an unknown number of lines of input from **stdin** (System.in) until reaching the end of file (EOF), you can use a **Scanner** object and a while loop. The loop will continue reading input as long as there is more input available.

Here's an example code snippet that demonstrates reading input lines until reaching EOF:

```java
import java.util.Scanner;

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

       while (scanner.hasNextLine()) {
          String line = scanner.nextLine();
          // Process each line of input here
          System.out.println("Input line: " + line);
       }

       scanner.close();
    }
}
```

In this code, the **hasNextLine()** method of the **Scanner** object is used to check if there is another line of input available. If there is, the **nextLine()** method is called to read the entire line as a **String**. You can process each line of input within the loop according to your requirements.

The loop will continue until there are no more lines of input available, indicating that the end of file (EOF) has been reached.

Note that after finishing reading input, it's good practice to close the **Scanner** object using the **close()** method to release any associated resources.

Feel free to modify the code inside the loop to perform the desired operations on each line of input.

BufferReader

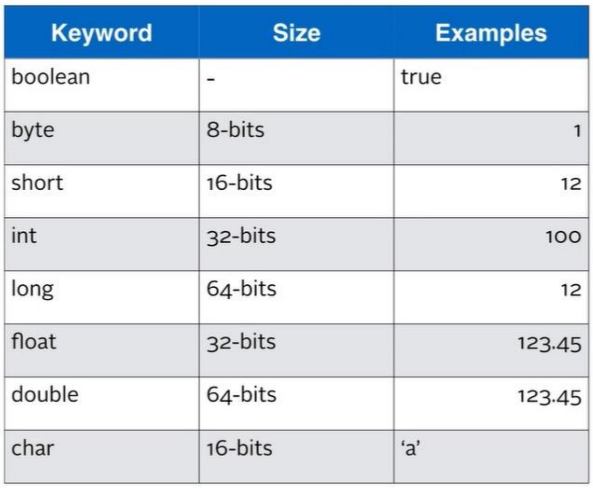
```
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(Sy
stem.in));


        int N = Integer.parseInt(bufferedReader.readLine().trim());

        bufferedReader.close();
```

Range of primitive data types

Java has 8 primitive data types; char, boolean, byte, short, int, long, float, and double. For this exercise, we'll work with the primitives used to hold integer values (byte, short, int, and long):

- A byte is an 8-bit signed integer. -128 to 127
- A short is a 16-bit signed integer. -32,768 to 32,767
- An int is a 32-bit signed integer. -2,147,483,648 to 2,147,483,647
- A long is a 64-bit signed integer. -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

An 8-bit signed integer, also known as a byte in Java, can store values in the range from -128 to 127. The most significant bit (MSB) is reserved for the sign (0 for positive, 1 for negative), and the remaining 7 bits are used to represent the magnitude of the value.



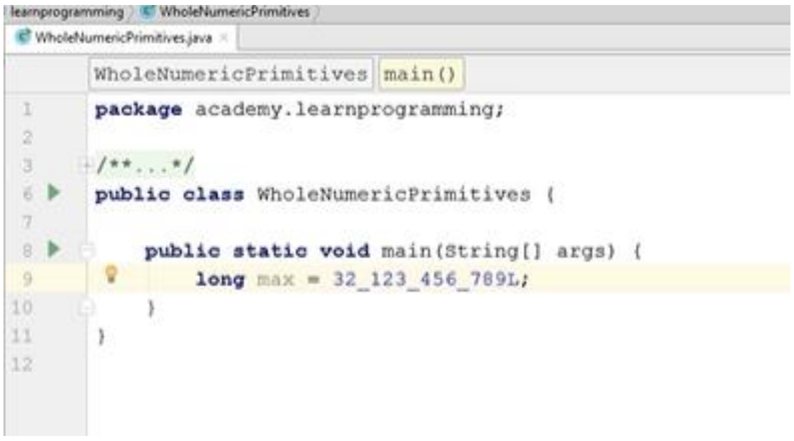
Keyword	Size	Examples
boolean	-	true
byte	8-bits	1
short	16-bits	12
int	32-bits	100
long	64-bits	12
float	32-bits	123.45
double	64-bits	123.45
char	16-bits	‘a’

Assigning literals to variables

In java number literals are by defaults read as integers. So to assign a value to any other data type or to represent them in other formats we need to use appropriate letters.

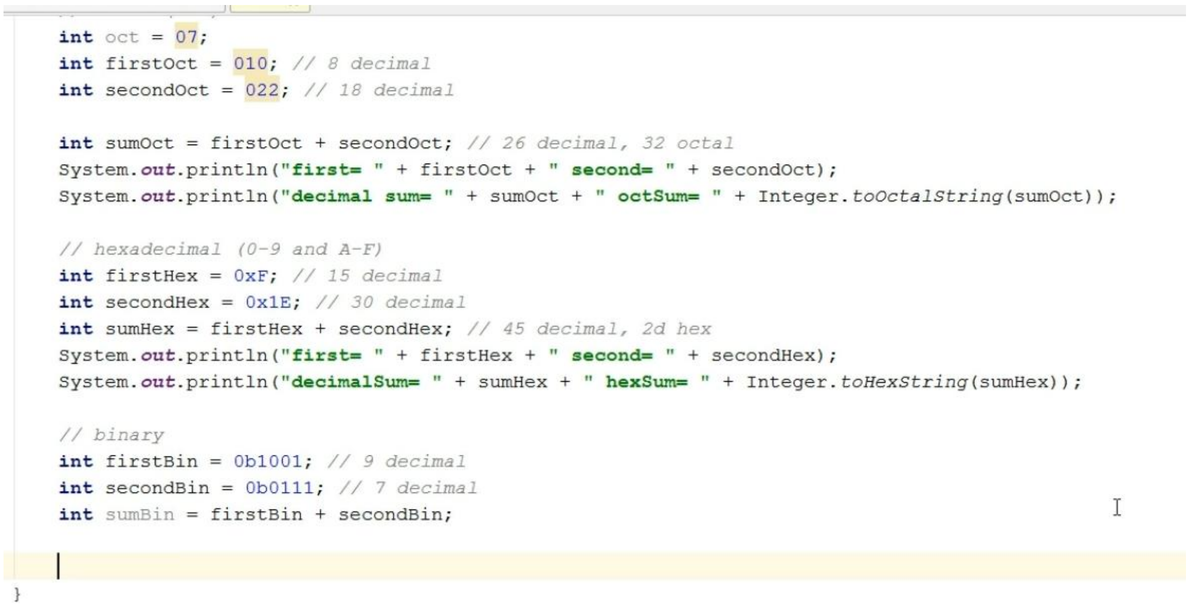
For example :

- L or l for long
- F or f for floating point number



```java
package academy.learnprogramming;

/**...*/
public class WholeNumericPrimitives {

    public static void main(String[] args) {
        long max = 32_123_456_789L;
    }
}
```

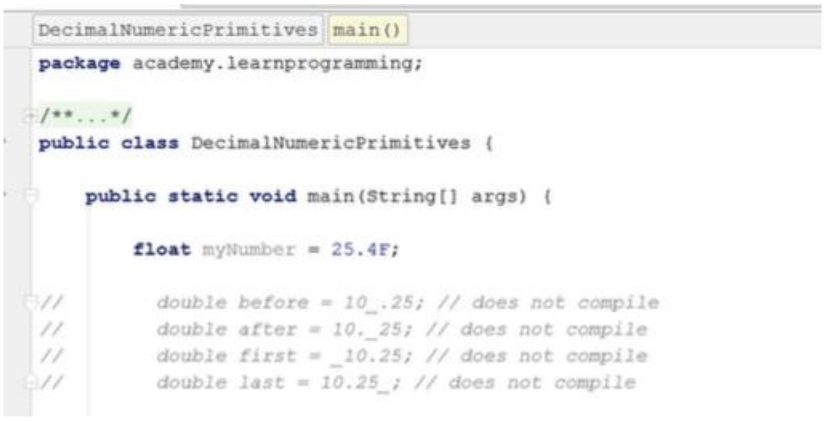
Representation of octal, hexadecimal and binary number



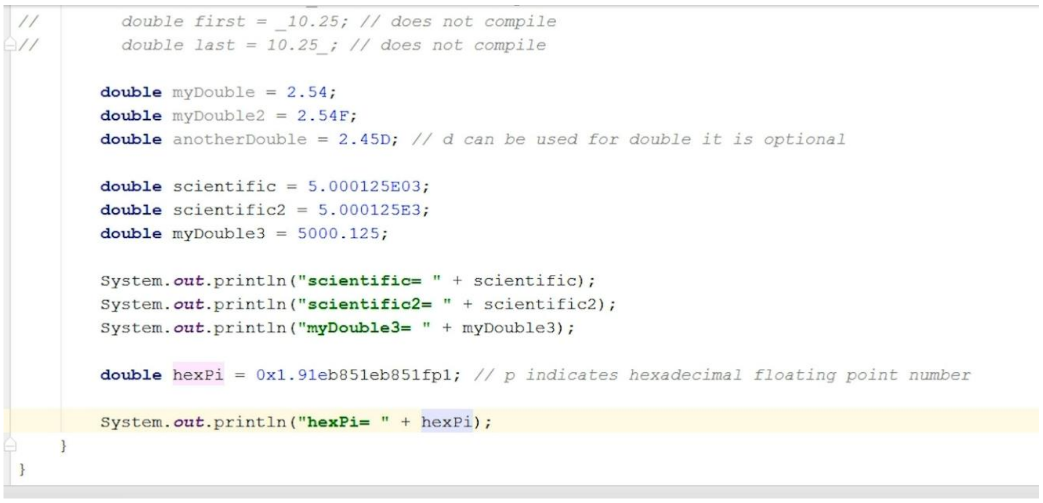
```java
    int oct = 07;
    int firstOct = 010; // 8 decimal
    int secondOct = 022; // 18 decimal

    int sumOct = firstOct + secondOct; // 26 decimal, 32 octal
    System.out.println("first= " + firstOct + " second= " + secondOct);
    System.out.println("decimal sum= " + sumOct + " octSum= " + Integer.toOctalString(sumOct));

    // hexadecimal (0-9 and A-F)
    int firstHex = 0xF; // 15 decimal
    int secondHex = 0x1E; // 30 decimal
    int sumHex = firstHex + secondHex; // 45 decimal, 2d hex
    System.out.println("first= " + firstHex + " second= " + secondHex);
    System.out.println("decimalSum= " + sumHex + " hexSum= " + Integer.toHexString(sumHex));

    // binary
    int firstBin = 0b1001; // 9 decimal
    int secondBin = 0b0111; // 7 decimal
    int sumBin = firstBin + secondBin;

}
```

- Octal are represented with adding a 0 at the beginning
- Hexadecimal are represented with adding a 0x at the beginning
- Binary are represented with adding a 0b at the beginning

Representation of decimal numbers



```
DecimalNumericPrimitives  main()
package academy.learnprogramming;

/**...*/
public class DecimalNumericPrimitives {

    public static void main(String[] args) {

        float myNumber = 25.4F;

//          double before = 10_.25; // does not compile
//          double after = 10._25; // does not compile
//          double first = _10.25; // does not compile
//          double last = 10.25_; // does not compile
```



```
//          double first = _10.25; // does not compile
//          double last = 10.25_; // does not compile

        double myDouble = 2.54;
        double myDouble2 = 2.54F;
        double anotherDouble = 2.45D; // d can be used for double it is optional

        double scientific = 5.000125E03;
        double scientific2 = 5.000125E3;
        double myDouble3 = 5000.125;

        System.out.println("scientific= " + scientific);
        System.out.println("scientific2= " + scientific2);
        System.out.println("myDouble3= " + myDouble3);

        double hexPi = 0x1.91eb851eb851fp1; // p indicates hexadecimal floating point number

        System.out.println("hexPi= " + hexPi);
    }
}
```

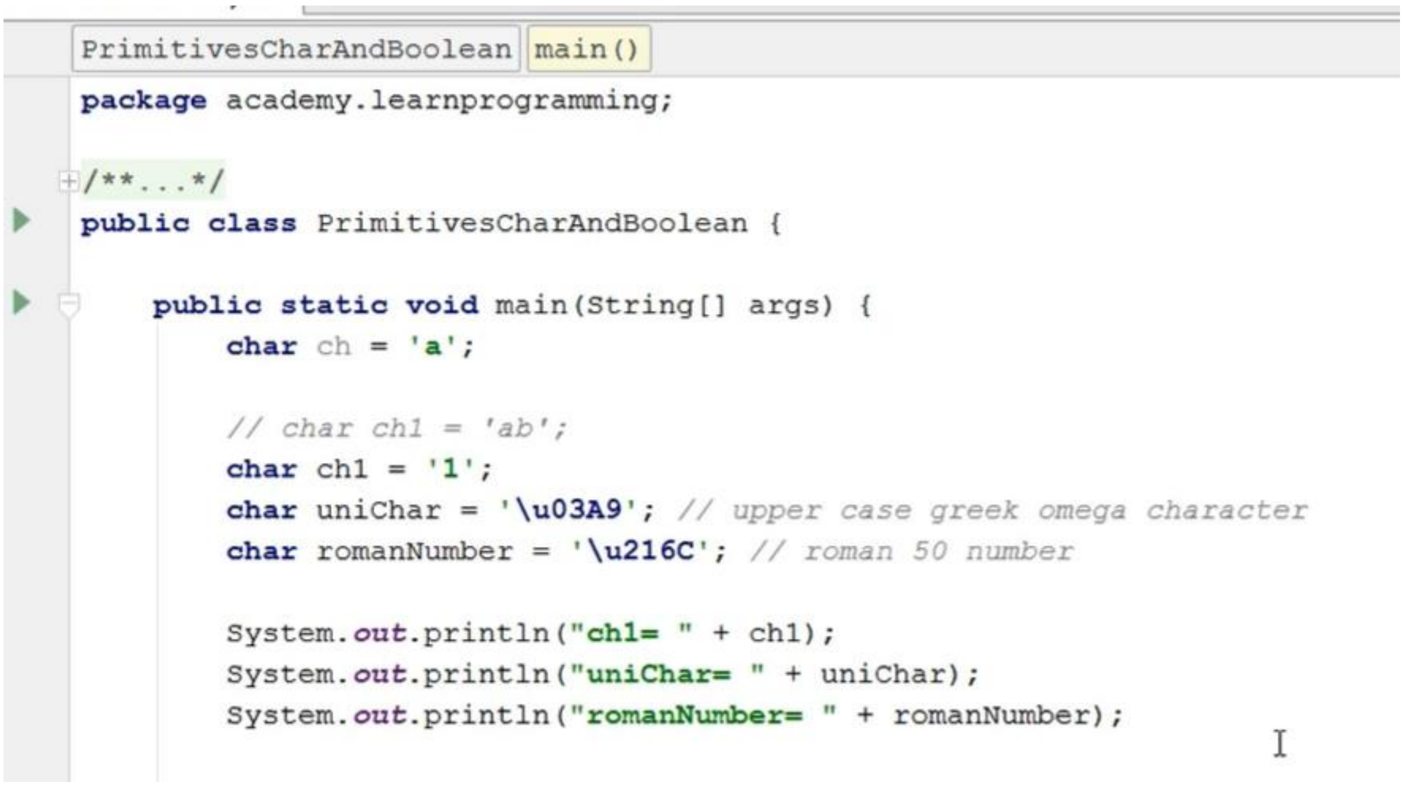
Hexadecimal
The value 0x1.91ebfp1 is a hexadecimal floating-point literal in Java. Java allows you to write floating-point numbers in hexadecimal notation for precise representation of binary fractions.

Here's a breakdown of the hexadecimal floating-point literal 0x1.91ebfp1:

- 0x: Indicates that the number is in hexadecimal format.
- 1.91ebf: The hexadecimal digits of the number.
  - 1: The integral part.
  - .91ebf: The fractional part.
- p1: The exponent part, which is in base 2 (binary). The p is similar to e in decimal scientific notation, but it signifies a power of 2.

Decimal
for simple base 10 number we use e or E followed by the exponent power as shown in the figure.

Primitive char and Unicode characters



```
package academy.learnprogramming;

/**...*/
public class PrimitivesCharAndBoolean {

    public static void main(String[] args) {
        char ch = 'a';

        // char ch1 = 'ab';
        char ch1 = '1';
        char uniChar = '\u03A9'; // upper case greek omega character
        char romanNumber = '\u216C'; // roman 50 number

        System.out.println("ch1= " + ch1);
        System.out.println("uniChar= " + uniChar);
        System.out.println("romanNumber= " + romanNumber);
```

Default Values Assigned to Primitive Data Types

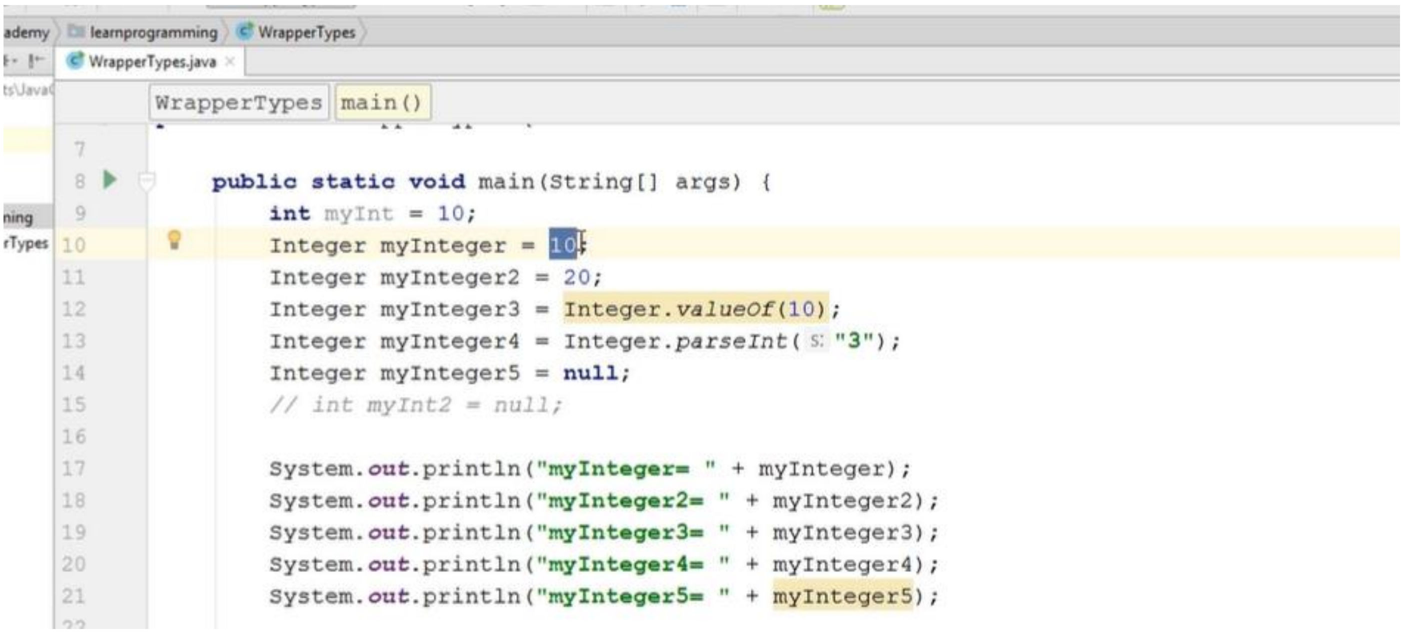
In Java, when a variable is declared but not initialized, it is assigned a default value based on its data type. The default values for the primitive data types in Java are as follows:

byte: 0
short: 0
int: 0
long: 0L
float: 0.0f
double: 0.0d
char: ‘\u0000’ (null character)
boolean: false

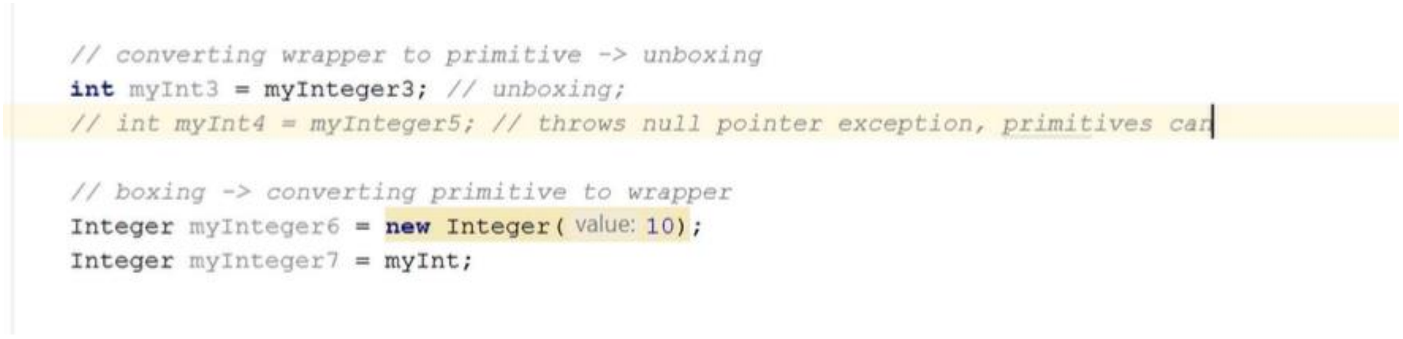
It is important to note that these default values are only assigned if the variable is not explicitly initialized with a value. If a variable is initialized with a value, that value will be used instead of the default.

# It only provides this default value initialization for class variables not for local variables

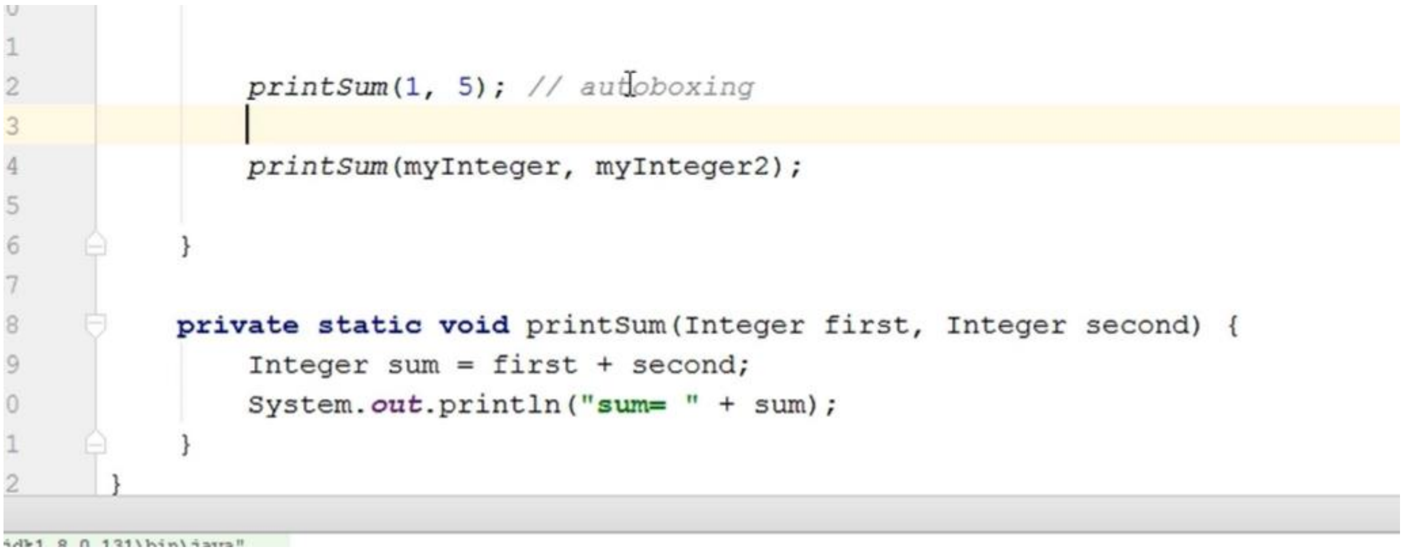
Wrapper Classes



```java
    public static void main(String[] args) {
        int myInt = 10;
        Integer myInteger = 10;
        Integer myInteger2 = 20;
        Integer myInteger3 = Integer.valueOf(10);
        Integer myInteger4 = Integer.parseInt( s: "3");
        Integer myInteger5 = null;
        // int myInt2 = null;

        System.out.println("myInteger= " + myInteger);
        System.out.println("myInteger2= " + myInteger2);
        System.out.println("myInteger3= " + myInteger3);
        System.out.println("myInteger4= " + myInteger4);
        System.out.println("myInteger5= " + myInteger5);
```



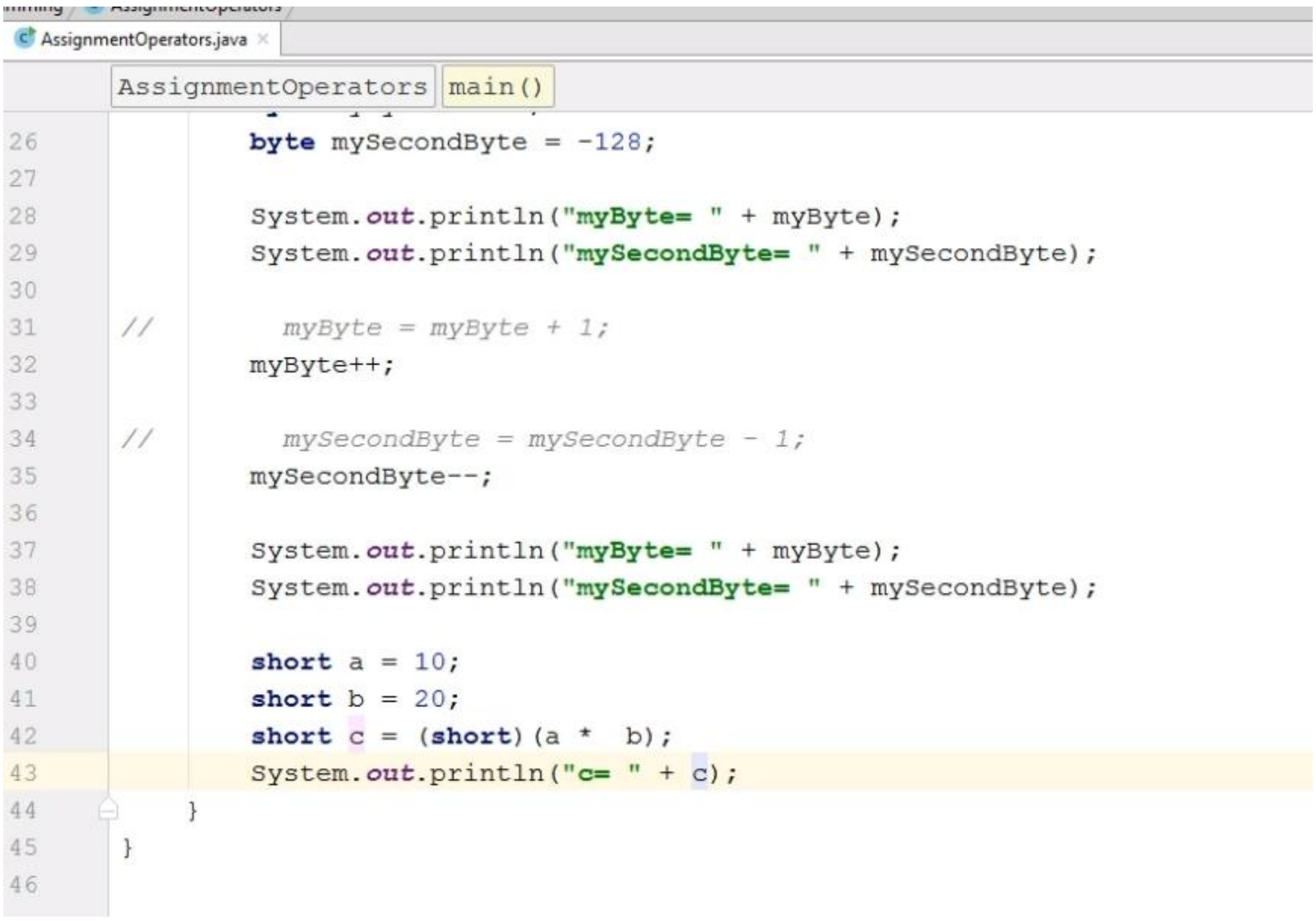
```java
    // converting wrapper to primitive -> unboxing
    int myInt3 = myInteger3; // unboxing;
    // int myInt4 = myInteger5; // throws null pointer exception, primitives can

    // boxing -> converting primitive to wrapper
    Integer myInteger6 = new Integer( value: 10);
    Integer myInteger7 = myInt;
```



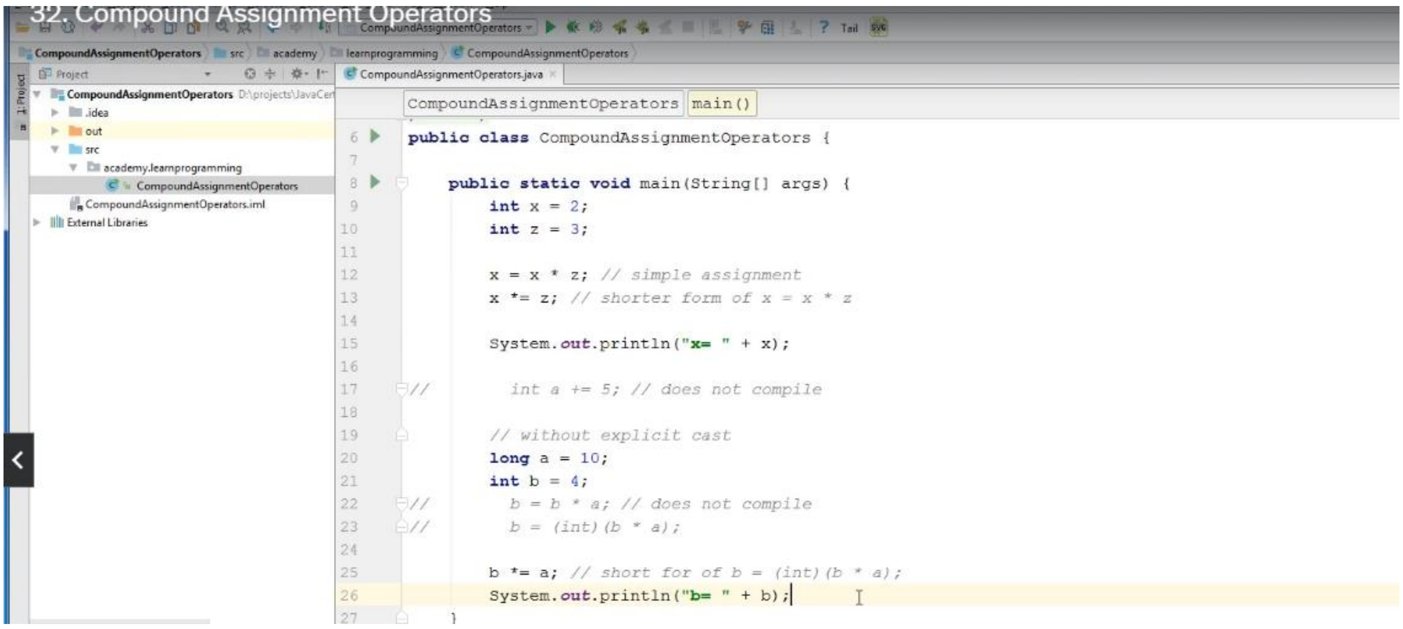
```java

        printSum(1, 5); // autoboxing

        printSum(myInteger, myInteger2);

    }

    private static void printSum(Integer first, Integer second) {
        Integer sum = first + second;
        System.out.println("sum= " + sum);
    }
}
```

Boxing and Autoboxing

Assignment operator

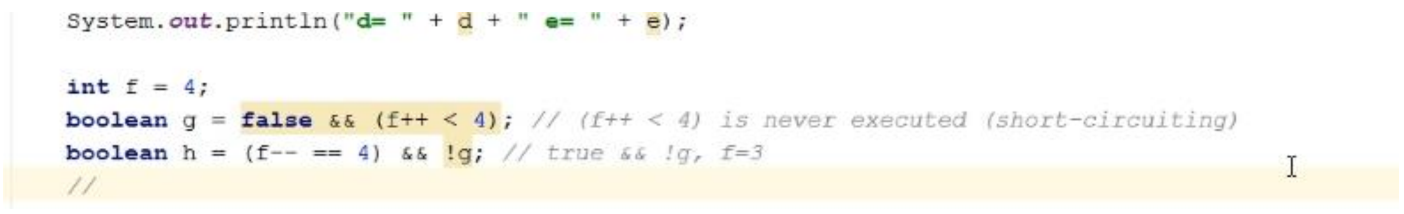


```java
        byte mySecondByte = -128;

        System.out.println("myByte= " + myByte);
        System.out.println("mySecondByte= " + mySecondByte);

//        myByte = myByte + 1;
        myByte++;

//        mySecondByte = mySecondByte - 1;
        mySecondByte--;

        System.out.println("myByte= " + myByte);
        System.out.println("mySecondByte= " + mySecondByte);

        short a = 10;
        short b = 20;
        short c = (short) (a *  b);
        System.out.println("c= " + c);
    }
}
```

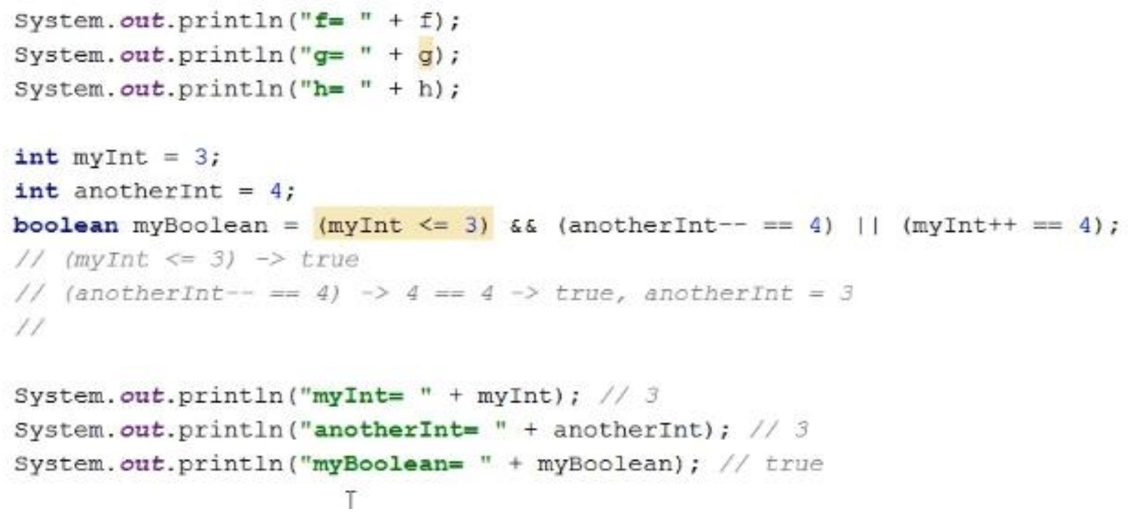

32. Compound Assignment Operators
CompoundAssignmentOperators
src
academy
learnprogramming
CompoundAssignmentOperators
Project
CompoundAssignmentOperators D:\projects\JavaCer
.idea
out
src
academy.learnprogramming
CompoundAssignmentOperators
CompoundAssignmentOperators.iml
External Libraries
CompoundAssignmentOperators.java
main()

```java
public class CompoundAssignmentOperators {

    public static void main(String[] args) {
        int x = 2;
        int z = 3;

        x = x * z; // simple assignment
        x *= z; // shorter form of x = x * z

        System.out.println("x= " + x);

//          int a += 5; // does not compile

        // without explicit cast
        long a = 10;
        int b = 4;
//          b = b * a; // does not compile
//          b = (int) (b * a);

        b *= a; // short for of b = (int) (b * a);
        System.out.println("b= " + b);
    }
```

Short circuiting



```
System.out.println("d= " + d + " e= " + e);

int f = 4;
boolean g = false && (f++ < 4); // (f++ < 4) is never executed (short-circuiting)
boolean h = (f-- == 4) && !g; // true && !g, f=3
//
```



```
System.out.println("f= " + f);
System.out.println("g= " + g);
System.out.println("h= " + h);

int myInt = 3;
int anotherInt = 4;
boolean myBoolean = (myInt <= 3) && (anotherInt-- == 4) || (myInt++ == 4);
// (myInt <= 3) -> true
// (anotherInt-- == 4) -> 4 == 4 -> true, anotherInt = 3
//

System.out.println("myInt= " + myInt); // 3
System.out.println("anotherInt= " + anotherInt); // 3
System.out.println("myBoolean= " + myBoolean); // true
```

anotherInt will remain 3 because of short circuiting

== and equals method

Int x = 4;

Int y = 4;

Boolean c = x==y; //c = true

Integer x = 4;

Integer y = 4;

Boolean c = x==y // c = true;

Integer x = 128;

Integer y = 128;

Boolean c = x == y // c = false;

Integer x = 4;

Integer y = new Integer(4);

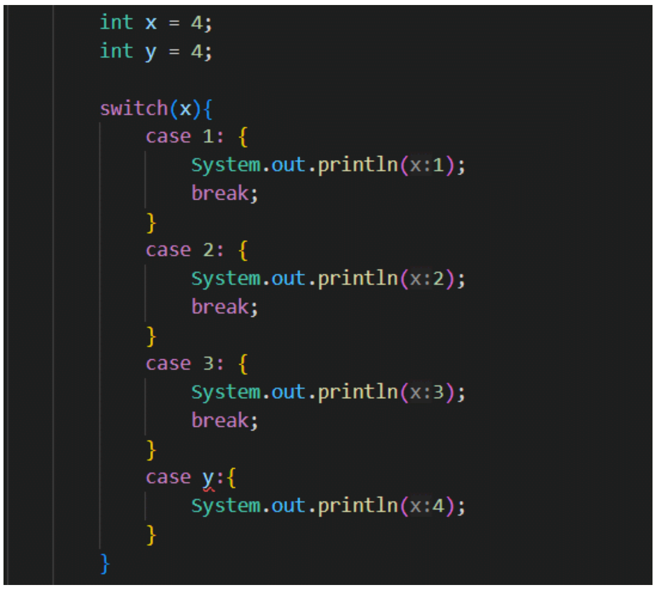
Boolean c = x==y // c = false;

- The equals operator == when used with primitive data type compares the value stored at the memory location of the variable.
- The equals operator == when used with objects (here of type Integer) then is compares the hash value of the memory location for the object.
- Now it returns true in case of Integer object value from -128 to 127 because there is a cache in Integer wrapper class that gives same memory location to each object made for the same numerical value in this range.
- For numeric values outside this range it creates a new instance of object.
- But if you create an object with the new keyword then no matter the numeric value it creates a new instance of the object.

The switch case conflict

The Java language specification does not allow non-constant variables in switch case labels. This is because the case labels must be compile-time constants, and non-final variables are not considered compile-time constants.

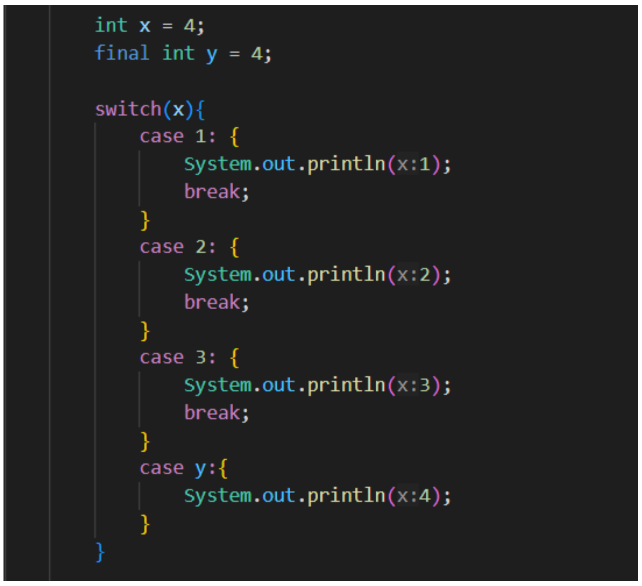
Hence:



```
int x = 4;
int y = 4;

switch(x){
    case 1: {
        System.out.println(x:1);
        break;
    }
    case 2: {
        System.out.println(x:2);
        break;
    }
    case 3: {
        System.out.println(x:3);
        break;
    }
    case y:{
        System.out.println(x:4);
    }
}
```

The above code shows compilation error

while:



```
int x = 4;
final int y = 4;

switch(x){
    case 1: {
        System.out.println(x:1);
        break;
    }
    case 2: {
        System.out.println(x:2);
        break;
    }
    case 3: {
        System.out.println(x:3);
        break;
    }
    case y:{
        System.out.println(x:4);
    }
}
```

This code above doesn’t

this happens because in the first one y is just a normal integer variable and in the second one it is declared as final integer variable.

Static initialization blocks

Static initialization blocks in Java are used to initialize static variables or perform other initialization tasks for a class. They are executed when the class is loaded into the Java Virtual Machine (JVM) and only once, regardless of how many instances of the class are created.

Here's the general syntax of a static initialization block:

```
        static { // Initialization code }
```

The static initialization block is defined using the **static** keyword followed by a pair of curly braces. Inside the block, you can write any valid Java code, including variable assignments, method calls, or other initialization tasks.

Key points to understand about static initialization blocks:

1. Execution Timing: The static initialization block is executed when the class is first loaded into the JVM, before any static variables or static methods of the class are accessed or invoked.
2. Order of Execution: If a class has multiple static initialization blocks, they are executed in the order they appear in the code.
3. Access to Static Members: Inside the static initialization block, you can access and modify static variables and invoke static methods of the class freely. However, you cannot access instance variables or instance methods directly since they are associated with specific object instances.
4. Exception Handling: If an exception occurs during the execution of the static initialization block, the class fails to initialize, and an exception is thrown. This exception can be caught and handled using standard Java exception handling mechanisms.

Static initialization blocks are useful when you need to perform complex initialization logic for static variables or execute additional tasks during class loading. They are commonly used to initialize static variables based on calculations or external data sources, set up static data structures, or establish connections to external resources.

Here's a simple example demonstrating the usage of a static initialization block:

```
public class MyClass {
    static int myStaticVariable;

    static {
        // Perform initialization
        myStaticVariable = 42;
        System.out.println("Static initialization block executed");
    }

    public static void main(String[] args) {
        System.out.println("Value of myStaticVariable: " + myStaticVariable);
    }
}
```

In this example, the static initialization block sets the value of **myStaticVariable** to 42. When the class **MyClass** is loaded, the static initialization block is executed, and the message "Static initialization block executed" is printed. Later, in the **main** method, the value of **myStaticVariable** is printed, showing the initialized value.

Static initialization blocks are a powerful feature in Java that allow you to perform necessary setup tasks when a class is loaded. They provide flexibility and help ensure proper initialization of static variables and resources.

To put constraints on the input while using the static initialization block, you can add conditional statements within the block to validate the input and handle any invalid inputs accordingly.

Here's an example:

```java
import java.util.Scanner;

public class MyClass {
    static int myStaticVariable;

    static {
       Scanner scanner = new Scanner(System.in);

       // Read input until a valid value is provided
       while (true) {
          System.out.print("Enter a positive integer: ");
          if (scanner.hasNextInt()) {
             int input = scanner.nextInt();
             if (input > 0) {
                myStaticVariable = input;
                break; // Valid input, exit the loop
             } else {
                System.out.println("Invalid input! Please enter a positive integer.");
             }
          } else {
             System.out.println("Invalid input! Please enter an integer.");
             scanner.next(); // Consume the invalid input
          }
       }

       scanner.close();
       System.out.println("Static initialization block executed");
    }

    public static void main(String[] args) {
       System.out.println("Value of myStaticVariable: " + myStaticVariable);
    }
}
```

Strings

Strings are immutable so any of the below methods will not make any changes to which they are called for instead they will return a new string. (Except the length and char that will return int and char respectively)

length()

charAt()

indexOf()

Syntax: indexOf(ch, fromindex) // fromindex parameter is optional

String str = “java is fun”;

str.indexOf(‘a’); // 1

str.indexOf(‘a’, 2); //4

str.indexOf(‘fun’); //8

str.indexOf(‘fun’,10); //-1 match wasn’t found

substring()

Syntax: substring(beginIndex, endIndex) // endIndex is optional

// beginIndex is inclusive and endIndex is excalusive

// end parameter should always be greater than start parameter.

Str.substring(4,4) // will return empty string

str.substring(4,2) //string index out of range

toUpperCase()

Syntax : str.toUpperCase();

System.out.println(“abc”.toUpperCase()); // this is a valid syntax

toLowerCase()

equals()

Sytax: str1.equals(str2); // case sensitive

equalsIgnoreCase()

ignores case.

startsWith()

Syntax: str1.startsWith(str2) || str1.startsWith(“literal”); // case sensitive

endsWith()

//case sensitive

contains()

//case sensitive

replace()

Syntax : str1.replace(toReplace, replaceWith);

String myString = “Java is cool”;

System.out.println(myString.replace(‘a’, ‘A’);

trim()

Syntax: str.trim()

Trims any spaces in the beginning or ending of a string.

System.out.println(“ Java “.trim()); // will remove spaces before and after java

System.out.println(“ Java is cool “.trim()); // will remove spaces before and after whole string Java is cool and not from between.

Chaining methods

String str = “ Java “.trim().toLowerCase().replace(‘j’,’J’); // output = Java

// in this first “ Java “.trim() will return “Java” then “Java”.toLowerCase() will return java and so on

String to int and int to String

- The Integer.parseInt(s) method is a built-in Java method that parses a String representation of an integer and converts it into an int value. It takes a String parameter s and returns an int value that represents the parsed integer.
- To convert an int to a String in Java, you can use the String.valueOf() method or concatenate the int value with an empty String.
- Concatenating with an empty String

Concatenating with an empty String:

```
int number = 42;
String numberString = "" + number;
```

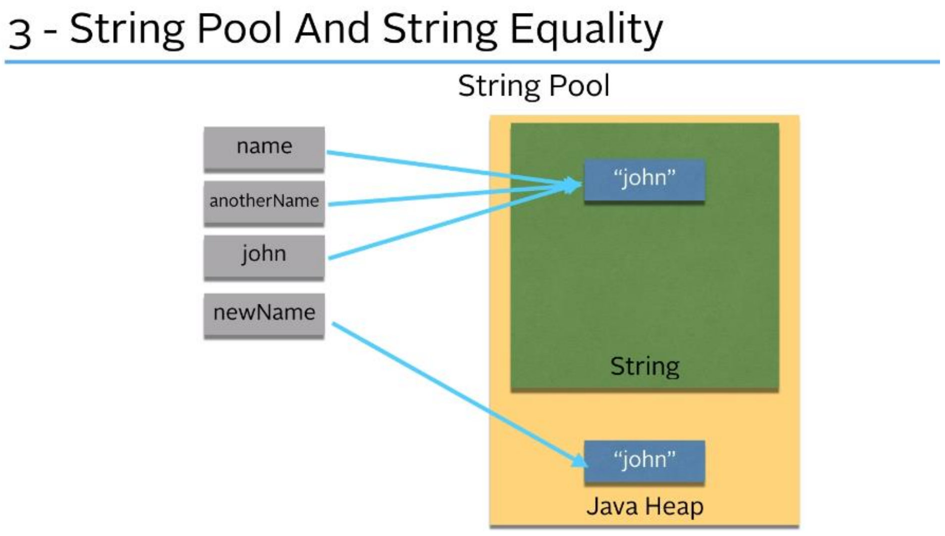
Integer.parseInt()

Converts string to primitive int

Integer.valueOf()

Converts string to wrapper class Integer

String pool and string Equality



When we create a string in java using the String class then it checks if the same string is available in the string pool and if it is then the new string just points to that same string in the java string pool
But if we use the new keyword to create a string then in that case java creates a new instance of the string in the java heap.

so a string

String name = “John”

String name2 = “John”

String name3 = new String(“John”)

Boolean isEqual;

isEqual = name == name2 // true;

isEqual = name == name3 // false

but if we use the equals() method then it will return true in both cases as it checks the actual value of the string and not the hash value

Constant folding at compile time

When you use:

```
String str3 = "ab" + "c";
```

- Both `"ab"` and `"c"` are string literals.
- The Java compiler recognizes this and concatenates them at compile time, resulting in the string literal `"abc"`.
- Since `"abc"` is a string literal, it is interned and stored in the string pool.

Thus the code essentially becomes:

```
String str3 = "abc";
```

Both `str1` and `str3` refer to the same string literal in the string pool.

Runtime Concatenation

In contrast when you do:

```
String str3 = str2 + "c";
```

str2 is a variable, and "c" is a string literal.

The concatenation happens at runtime, resulting in a new string object that is not automatically interned.

Hence

```
String str1 = "abc";         // String literal "abc", stored in the string pool
String str2 = "ab";          // String literal "ab", stored in the string pool
String str3 = str2 + "c";    // Concatenation happens at runtime, creating a new string
object

boolean isEqual = str1 == str3;  // This will be false
```

StringBuilder

• **Purpose**:

- `StringBuilder` is used to create mutable (modifiable) strings.
- It is more efficient than `String` when performing multiple modifications, like appending, inserting, or deleting characters.

• **Key Characteristics**:

- **Mutable**: Unlike `String`, which creates a new object every time it's modified, `StringBuilder` modifies the object itself.
- **Not thread-safe**: It is not synchronized, so it's not safe for use in multi-threaded environments (use `StringBuffer` for thread safety).

• **Common Methods**:

- charAt(), indexOf(), length(), substring()
- append(String str): Adds the given string to the end.
- insert(int offset, String str): Inserts the string at the specified position.
- delete(int start, int end): Removes characters from the start index until before the end index. End index is excluded.
- deleteCharAt(int index): delete the character at the given index.
- reverse(): Reverses the sequence of characters.
- toString(): Converts the StringBuilder content to a String.

• **Performance**:

- More efficient than `String` for concatenation because it doesn't create new objects during modification.
- Ideal for scenarios where many string manipulations are required.

• **Example Usage**:

```
java
Copy code
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");      // Appends " World"
sb.insert(6, "Java ");    // Inserts "Java " at index 6
sb.reverse();             // Reverses the string
String result = sb.toString();  // Converts to String
```

• **Constructors**:

- `StringBuilder()`: Creates an empty `StringBuilder`.
- `StringBuilder(String str)`: Creates a `StringBuilder` with the specified string.
- `StringBuilder(int capacity)`: Creates a `StringBuilder` with the specified initial capacity.

• **Capacity**:

- The capacity of `StringBuilder` grows automatically as needed, but you can specify it manually for performance optimization.

Capacity vs. Length in `StringBuilder`

- **Length**:
  - Refers to the number of characters currently stored in the `StringBuilder`.
  - It is the size of the actual content.
  - Use `length()` method to get the length.

  Example:

```
java
Copy code
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.length());  // Output: 5
```

- **Capacity**:
  - Refers to the total allocated space for characters in the `StringBuilder`.
  - This is the maximum number of characters it can hold before needing to resize.
  - Use `capacity()` method to get the capacity.
  - The default capacity is 16 characters unless specified.

  Example:

```
java
Copy code
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.capacity());  // Output: 21 (5 + 16 extra capacity)
```

- **Capacity Resizing**:
  - When the number of characters exceeds the current capacity, `StringBuilder` automatically increases the capacity (usually doubling it).
  - You can also set the capacity manually using the constructor or `ensureCapacity()` method.

Ways to Specify Capacity:

1. **Using Constructor**:
   - You can specify the initial capacity of a `StringBuilder` when creating it.

```
java
Copy code
StringBuilder sb = new StringBuilder(50); // Initializes with a capacity of 50
```

2. **Using `ensureCapacity(int minimumCapacity)`**:
   - You can use the `ensureCapacity` method to ensure the `StringBuilder` has at least the specified capacity.

```
java
Copy code
StringBuilder sb = new StringBuilder();
sb.ensureCapacity(100); // Ensures capacity is at least 100
```

If you initialize a `StringBuilder` with the default capacity of 16 and keep appending characters, once you append the 17th character, the `StringBuilder` will automatically increase its capacity.

What Happens Internally:

1. The initial capacity of a `StringBuilder` is 16 (if no capacity is specified).
2. When you append characters and the total length exceeds the current capacity (16 in this case), the `StringBuilder` automatically increases its capacity.
3. The new capacity is typically **(current capacity * 2) + 2**. So, after reaching 16 characters, the capacity will grow to 34.

Why Strings are immutable in java?

Strings in Java are immutable for several important reasons, primarily focused on security, performance, and memory efficiency. Here's a breakdown of why Java makes strings immutable:

1. Security:

- **Immutable strings prevent data tampering**: In Java, strings are commonly used for sensitive data, such as usernames, passwords, network connections, and file paths. If strings were mutable, a malicious actor could modify the string's content, leading to security vulnerabilities.
- **Thread safety**: Since immutable objects cannot be modified after creation, they are inherently thread-safe. Multiple threads can use the same string without needing synchronization, avoiding race conditions or other concurrency issues.

2. Caching and Performance (String Pool):

- **String interning**: Java maintains a **string pool**, where strings with the same value are stored only once. This is possible because strings are immutable. If strings were mutable, changing one string's value could unintentionally modify the same string in another part of the program, breaking the pooling mechanism.
- **Memory efficiency**: By storing only one instance of each string literal in the string pool, Java reduces memory consumption, as identical strings can share the same memory.

3. Hashcode Consistency:

- **Immutable strings ensure consistent hash codes**: In Java, strings are often used as keys in collections like `HashMap` and `HashSet`. The immutability of strings guarantees that once a string is created, its hash code will not change, ensuring it remains correctly located in hash-based data structures. If strings were mutable, their hash codes could change, making it impossible to retrieve values stored using the original hash code.

4. Design Simplicity:

- **Immutable objects are simpler to understand and use**: With immutability, you don't have to worry about unintended changes to a string. Once a string is created, it stays the same throughout the program, making debugging and reasoning about code easier.

5. Security in Class Loading and Reflection:

- Java uses strings for loading classes and for reflection operations. If strings were mutable, someone could tamper with the names of classes or methods, leading to unpredictable behavior or security

Date and time

There are three date and time class

- LocalDate
- LocalTime
- LocalDateTime

Date classes are immutable.

Methods

now() : LocalDate.now(), LocalTime.now(), LocalDateTime.now()

BigDecimal

The **BigDecimal** class in Java is a part of the **java.math** package and provides arbitrary-precision decimal arithmetic. It is used for precise numerical calculations where accuracy is critical, especially when dealing with large numbers or when the precision of floating-point numbers is insufficient.

Using **BigDecimal** can help avoid common pitfalls and inaccuracies associated with floating-point arithmetic. It is particularly useful in financial calculations, currency conversions, and other scenarios where precision is essential.

Here's an example of using **BigDecimal** to perform precise arithmetic calculations:

```java
import java.math.BigDecimal;

public class BigDecimalExample {
    public static void main(String[] args) {
        BigDecimal num1 = new BigDecimal("10.5");
        BigDecimal num2 = new BigDecimal("3.2");

        // Addition
        BigDecimal sum = num1.add(num2);
        System.out.println("Sum: " + sum); // Output: 13.7

        // Subtraction
        BigDecimal difference = num1.subtract(num2);
        System.out.println("Difference: " + difference); // Output: 7.3

        // Multiplication
        BigDecimal product = num1.multiply(num2);
        System.out.println("Product: " + product); // Output: 33.6

        // Division
        BigDecimal quotient = num1.divide(num2, 2, BigDecimal.ROUND_HALF_UP);
        System.out.println("Quotient: " + quotient); // Output: 3.28
    }
```

In the line **BigDecimal quotient = num1.divide(num2, 2, BigDecimal.ROUND_HALF_UP);**, the **divide** method of the **BigDecimal** class is used to perform division between two **BigDecimal** objects (**num1** and **num2**).
The **divide** method takes three arguments:

1. Divisor (**num2**): This is the **BigDecimal** object that will be divided by the dividend (**num1**).
2. Scale: The scale determines the number of digits to the right of the decimal point in the result. In this case, **2** is passed as the scale, indicating that the result should have two decimal places.
3. Rounding Mode: The rounding mode specifies how the result should be rounded in case there is a need for rounding. In this case, **BigDecimal.ROUND_HALF_UP** is used as the rounding mode. It is one of the rounding modes provided by the **BigDecimal** class and rounds towards the nearest neighbor. If the number is exactly halfway between two neighbors, it rounds up.
The **divide** method then performs the division operation and returns a new **BigDecimal** object representing the quotient. This object is assigned to the variable **quotient**.
Here's an example to illustrate the use of **divide** with **scale** and **roundingMode**:

To divide BigDecimal by an integer…

```java
BigDecimal percentage = y.divide(BigDecimal.valueOf(100));
```

BigInteger

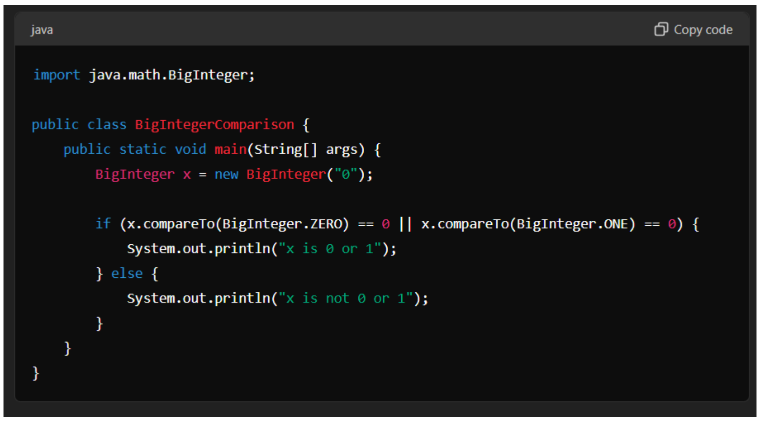
BigInteger is a class in the **java.math** package in Java. It is used for representing and performing arithmetic operations on integers with arbitrary precision. Unlike the primitive data types (**int**, **long**, etc.), **BigInteger** can handle integer values of any size, limited only by the amount of memory available.

Here are some key features and functionalities of **BigInteger**:

1. Arbitrary Precision: **BigInteger** can represent integers of any size, limited only by available memory. This makes it suitable for applications that require operations on extremely large numbers.
2. Immutable: **BigInteger** objects are immutable, meaning their values cannot be changed after they are created. Operations on **BigInteger** return new **BigInteger** objects rather than modifying the original object.
3. Arithmetic Operations: **BigInteger** provides methods for performing arithmetic operations such as addition, subtraction, multiplication, division, and modular arithmetic (remainder and modular exponentiation). These operations can be performed on **BigInteger** objects of different sizes.
4. Bitwise Operations: **BigInteger** supports bitwise operations like bitwise AND, bitwise OR, bitwise XOR, and bitwise shift operations.
5. Comparison and Equality: **BigInteger** provides methods for comparing **BigInteger** objects and checking for equality (**equals**) or inequality (**compareTo**).
6. Conversion: **BigInteger** can be converted to and from other primitive data types like **int**, **long**, **byte**, **short**, and vice versa.

To use **BigInteger**, you need to import the **java.math.BigInteger** class in your Java program. Here's an example of creating and performing arithmetic operations with **BigInteger**:

```java
import java.math.BigInteger;

public class BigIntegerExample {
    public static void main(String[] args) {
        BigInteger num1 = new BigInteger("12345678901234567890");
        BigInteger num2 = new BigInteger("98765432109876543210");

        BigInteger sum = num1.add(num2);
        BigInteger difference = num1.subtract(num2);
        BigInteger product = num1.multiply(num2);
        BigInteger quotient = num1.divide(num2);

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
        System.out.println("Product: " + product);
        System.out.println("Quotient: " + quotient);
    }
}
```



```java
import java.math.BigInteger;

public class BigIntegerComparison {
    public static void main(String[] args) {
        BigInteger x = new BigInteger("0");

        if (x.compareTo(BigInteger.ZERO) == 0 || x.compareTo(BigInteger.ONE) == 0) {
            System.out.println("x is 0 or 1");
        } else {
            System.out.println("x is not 0 or 1");
        }
    }
}
```

To check if all digits of a number are pairwise different

The statement **String.valueOf(x)** converts the integer **x** to its corresponding string representation. It is used to convert the number **x** into a string so that we can apply string operations on it.

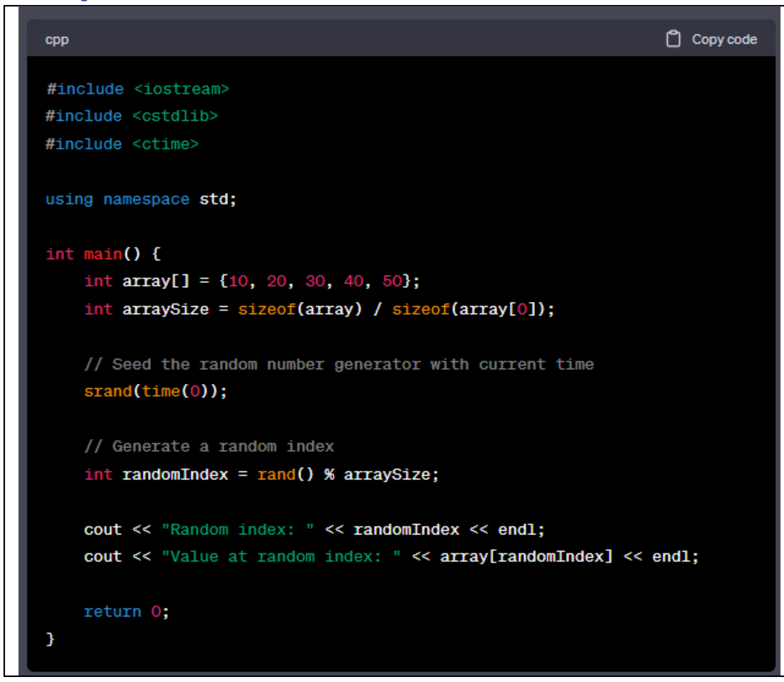
In the context of the code snippet, **isDistinct(String.valueOf(x))** is used to check if each digit of the number **x** is distinct. Here's how it works:

1. **String.valueOf(x)**: This converts the integer **x** to a string representation. For example, if **x** is **12345**, **String.valueOf(x)** will return the string **"12345"**. This allows us to access individual digits of the number.
2. **isDistinct(String.valueOf(x))**: This calls the **isDistinct** method with the string representation of **x** as the argument. The **isDistinct** method checks if each digit in the string is distinct. If all the digits are distinct, it returns **true**; otherwise, it returns **false**.

So, **isDistinct(String.valueOf(x))** is a way to check if each digit of the number **x** is

---

Why we use srand in c++;



```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    int array[] = {10, 20, 30, 40, 50};
    int arraySize = sizeof(array) / sizeof(array[0]);

    // Seed the random number generator with current time
    srand(time(0));

    // Generate a random index
    int randomIndex = rand() % arraySize;

    cout << "Random index: " << randomIndex << endl;
    cout << "Value at random index: " << array[randomIndex] << endl;

    return 0;
}
```

Seeding the random number generator with the current time is a common practice to make sure that the sequence of random numbers generated is different every time the program runs. The reason for this is that computers generate random numbers using deterministic algorithms, which means that if you start with the same initial conditions (known as the seed), you will get the same sequence of random numbers.
By using the current time as the seed, you introduce an element of unpredictability. Since the time changes every second, the seed will be different every time the program runs, leading to a different sequence of random numbers.

Abstract method

ABSTRACT CLASS

A class having atleast one abstract method

ABSTRACT METHOD

A method whose implementation is not defined - such methods are meant to be overridden

When an abstract class is subclassed, the subclass usually provides implementations for all of the methods in parent class. If it doesn’t, it must be declared abstract.

We cannot create object of an abstract class

Lambda Expression

A lambda expression is, essentially an anonymous(that is unnamed) method. However this method is not executed on its own. Instead it is used to implement a method defined by a functional interface.

Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as closures.

A functional interface is an interface that contains one and only one abstract method. Normally this methos specifies the intended purpose of the interface.

Interface

Interface variables are by default final static

Resources

https://skillcertpro.com/product/java-se-8-programmer-i-1z0-808-exam-questions/