**Java In One Short**

* **Type Conversion or Type Casting**

**Type Casting:** In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



**Types of Type Casting**

There are two types of type casting: Both types can support

1. **Widening Type Casting:** When we try to convert the small data type into the larger data type. **No, data loss** in this case. It also known as **implicit conversion** or **casting down**

**byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

1. **Narrowing Type Casting:** When we try to convert the large data type into the smaller data type. High **data loss** arises in this case. It also known as **explicit conversion** or **casting up**

**double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

* **Shadowing:**

It occurs when a variable declared within a certain scope (inner scope) has the same name as a variable declared in an outer scope (outer scope). The inner variable "shadows" the outer variable within its scope, making the outer variable temporarily inaccessible within that scope.  
  
**Example:**

public class ShadowingExample {

int x = 10; // Outer variable

public void shadowTest() {

int x = 20; // Inner variable, shadows the outer variable

System.out.println("Inner x: " + x); // Prints the value of inner x

// Access the outer x using 'this'

System.out.println("Outer x: " + this.x); // Prints the value of outer x

}

public static void main(String[] args) {

ShadowingExample example = new ShadowingExample();

example.shadowTest();

}

}

* **Varargs Arguments in methods(varags):**

In Java, varargs (variable-length arguments) allow you to create methods that can accept a variable number of arguments of the same type. This feature simplifies the syntax for methods that need to handle multiple arguments, especially when the number of arguments is not known at compile-time.

**Key Points about Varargs:**

**Syntax:**

1. Varargs are denoted by an ellipsis (...) following the type of the last parameter in a method signature.
2. The varargs parameter must be the last parameter in the method's parameter list.

**Example:**

returnType methodName(type... parameterName) {

// method body

}

**Note:** When we try varargs in function and not give any parameter in calling functions, then it return empty. When using varargs in method overloading, if you pass different arguments to each overloaded method, it resolves ambiguity effectively. However, if you do not pass any arguments to both overloaded methods, it may lead to ambiguity because the compiler cannot determine which method should be called, potentially resulting in a compilation error.

* **Strings and String Pool:**

String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The *java.lang.String* class is used to create a string object.

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use *StringBuffer* and StringBuilder classes.

There are two ways to create String object:

1. **By string literal**

Java String literal is created by using double quotes.

**Example:**

String s="welcome";

1. **By new keyword**

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String s=**new** String("Welcome");//creates two objects and one reference variable

**String Pool:**  
The string pool (or string constant pool) is a special memory area in the Java heap memory. It is used to store string literals and strings created using the String.intern() method. Here are some key points about the string pool:

**String Literal Pool**:

* When you create a string using a string literal (String str = "Hello";), Java checks if the string already exists in the pool.
* If it does, the existing string object is reused; if not, a new string object is created and placed in the pool.

String s1="Welcome";

String s2="Welcome"; //It doesn't create a new instance



**String Interning**:

* You can explicitly add a string to the pool using the intern() method. This method returns a canonical representation of the string object.

**Example:**

String str1 = new String("Hello").intern(); // Adds "Hello" to the pool

String str2 = "Hello"; // Reuses the existing "Hello" from the pool

1. **Benefits of String Pool**:

* **Memory Efficiency**: By reusing common strings, the string pool helps conserve memory.
* **Performance**: String comparison (using ==) is faster when comparing interned strings because they are stored in the same memory location.

1. **Considerations**:

* While the string pool is helpful, it's important to understand its behavior, especially when dealing with strings created dynamically (e.g., using the new keyword).

**Example of String Pool Usage:**

String str1 = "Hello"; // Created in the string pool

String str2 = "Hello"; // Reuses str1 from the string pool

String str3 = new String("Hello"); // Creates a new string object in the heap

String str4 = str3.intern(); // Adds "Hello" to the string pool and returns its reference

System.out.println(str1 == str2); // Output: true (same reference)

System.out.println(str1 == str3); // Output: false (different references)

System.out.println(str1 == str4); // Output: true (interned reference)

* **StringBuilder**

StringBuilder in Java is a class that provides a mutable sequence of characters. It is designed for efficient string manipulation operations, especially when you need to concatenate multiple strings or modify strings frequently. Here’s a detailed explanation of StringBuilder and why it is used:

**What is StringBuilder?**

1. **Mutable Nature**: Unlike the String class in Java, which is immutable (meaning once created, its value cannot be changed), StringBuilder is mutable. This means you can modify the contents of a StringBuilder object without creating a new instance every time you make a change.
2. **Efficiency**: StringBuilder is optimized for string concatenation and modification operations. When you concatenate strings using StringBuilder, it operates directly on the character array it holds internally, which avoids creating intermediate string objects. This results in better performance compared to repeated concatenation with the + operator or String.concat() method.
3. **Methods and Operations**: StringBuilder provides various methods to append, insert, delete, replace, and reverse strings efficiently. Some commonly used methods include:
   * **append(String str):** Appends the specified string to the end of the current sequence.
   * **insert(int offset, String str):** Inserts the specified string into the current sequence at the specified position.
   * **delete(int start, int end):** Removes characters from the current sequence between the specified start and end indices.
   * **replace(int start, int end, String str):** Replaces the characters in the current sequence between the specified start and end indices with the specified string.
   * **reverse():** Reverses the characters in the current sequence.

**Why Use StringBuilder?**

* **Concatenation Efficiency**: When you concatenate strings using StringBuilder, it avoids the overhead of creating and garbage collecting many intermediate String objects that can occur with repeated use of the + operator or String.concat().
* **Performance Benefits**: Especially in scenarios where strings are concatenated or modified within loops, StringBuilder provides significant performance improvements over using String.
* **Memory Efficiency**: By directly manipulating a mutable character array, StringBuilder reduces memory overhead compared to immutable String objects.
* **Convenience**: StringBuilder simplifies the process of dynamically constructing strings and modifying them in place, making code more readable and efficient.

**Example:**

StringBuilder sb = new StringBuilder();

// Append strings

sb.append("Hello");

sb.append(" ");

sb.append("world");

// Insert into StringBuilder

sb.insert(6, "Java "); // "Hello Java world"

// Replace characters

sb.replace(11, 16, "Java Programmer"); // "Hello Java Programmer"

// Delete characters

sb.delete(5, 11); // "Hello Programmer"

// Reverse the StringBuilder

sb.reverse(); // "remmargorP olleH"

// Convert StringBuilder to String

String result = sb.toString(); // "Programmer Hello"

* **Bitwise Operator**

1. **AND ( & ):** The bitwise AND operation **(&) is used with 1** to isolate the least significant bit (LSB) after shifting.
2. **OR ( | )**
3. **XOR ( ^ ) :** Performs a bitwise XOR between two numbers. For each bit position, the result is 1 if the bits are different, and 0 if they are the same.

* **Self-inverse property:** For any given number x, (x ^ x) = 0. This means if you XOR a number with itself, you get 0.
* **Zero property:** For any given number x, (x ^ 0) = x. This means if you XOR a number with 0, you get the number itself.
* **Commutative property:** For any two numbers x and y, (x ^ y) = (y ^ x). This means the order in which numbers are XORed does not matter.
* **Associative property:** For any three numbers x, y, and z, x ^ (y ^ z) = (x ^ y) ^ z. This means the way in which numbers are grouped in an XOR operation does not matter.

**Properties:**

* **a ^ 1** flips the least significant bit of **a**.

**Example:**  
a = 7 (which is `0111` in binary).

a ^ 1, we get `6` (which is `0110` in binary). As you can see, the least significant bit has been flipped.

* **a ^ 0** results in **a**, since XOR with 0 doesn't change the number.

**Example:**  
a = 9 (which is `1001` in binary).

a ^ 0, we get `9` (which is `1001` in binary). As you can see, the number remains the same.

* **a ^ a** result in **0**, because XORing a number with itself results in all bits being 0.

**Example:**  
a = 8 (which is `1000` in binary).

a ^ a (or `8 ^ 8`), we get `0`. As you can see, XORing a number with itself results in `0`.

**Example:**

Best example for XOR is, we use it for swapping two number and used for finding the unique element in the array if number is repeated n time and only one number is not repeated.

a = 1101 (which is 13 in decimal) and b = 1011 (which is 11 in decimal),

ans = a ^ b;

the XOR operation would give 0110 (which is 6 in decimal).

* **Masking**

In programming, a **mask** is a pattern of bits that you use in bitwise operations to preserve or change certain bits within a number. The process of applying a mask to a value is known as **masking**.

In Java, we can use bitwise operators (&,|, ^, ~, <<, >>, >>>) to perform masking operations.

**Note:** **Unsigned Right Shift Operator (>>>):**

The **>>>** operator in Java is known as the **unsigned right shift operator**. Unlike the `**>>**` (signed right shift) operator, which fills the leftmost bits with the sign bit (0 for positive numbers, 1 for negative numbers), the `**>>>**` operator always fills the leftmost bits with zeros, regardless of the sign of the number.

This operator is useful when you're dealing with the binary representations of numbers, and you don't want the sign of the number to influence the result of the shift.

**For example:** it can be used when you're treating `int` and `long` as 32- and 64-bit unsigned integral types, which are missing from the Java language.

int number = -4; // Binary: 11111111111111111111111111111100

int result = number >>> 1; // Binary: 01111111111111111111111111111110

In this example, the `>>>` operator shifts the bits of `number` one place to the right and fills the leftmost bit with `0`. The result is `2147483646` in decimal.

**Ques: Set the ith bit of number.**

**Discussion:** Setting the i-th bit of a number refers to changing the value of the i-th bit (0-based index)  
in the binary representation of that number to 1. If the bit is already 1, it remains unchanged.  
  
This operation is useful for manipulating individual bits in binary data.  
  
**Example 1:** n = 18 (in binary: 10010), i = 3 (set the 3 bit of number)   
 ans = 11010 (in decimal: 26)  
  
**Example 2:** n = 18 (in binary: 10010), i = 1 (set the 1 bit of number, it is already one so nothing change)  
 ans = 11010 (in decimal: 18)  
   
**Steps to solve this by maksing:**  
n = 10010, i = 3  
mask = 1 << 3 // 01000  
  
ans = n | mask   
 1 0 0 1 0  
 | 0 1 0 0 0  
 --------------  
 1 1 0 1 0  
 ---------------

**Example:**

private static int setithBitOfNumber(int n, int i) {  
 int mask = (1 << i);  
 return n | mask;  
}