**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)