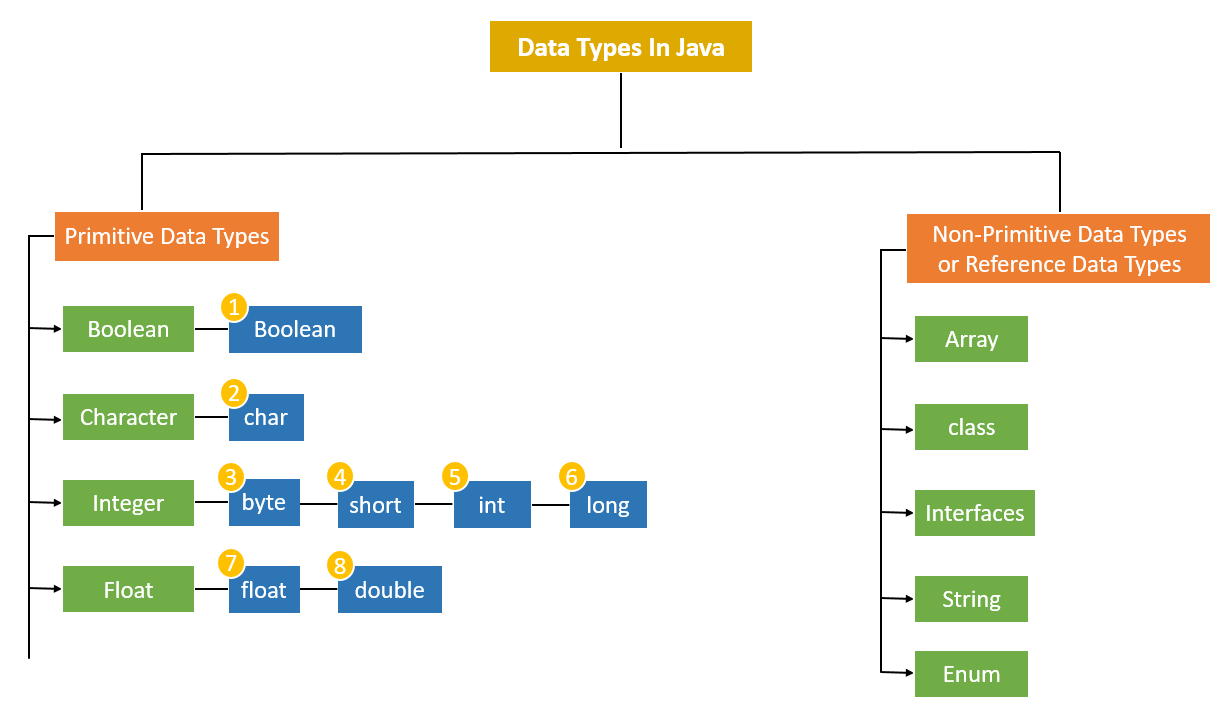
Primitive Data Types

A data type is a classification of data. It tells the compiler or interpreter how the programmer aims to use the variables or method. Data types are a crucial factor in all computer [programming](https://www.shiksha.com/online-courses/what-is-programming-st619) languages.

Data types in Java specify how memory stores the values of the variable. Each variable has a data type that decides the value the variable will hold. Moreover, Primitive Data Types are also used with functions to define their return type.

Data types in Java are divided into 2 categories:

* [**Primitive Data Types**](https://www.shiksha.com/online-courses/articles/data-types-in-java-primitive-and-non-primitive-data-types/#p1)
* [**Non-Primitive Data Types**](https://www.shiksha.com/online-courses/articles/data-types-in-java-primitive-and-non-primitive-data-types/#p2)



**Primitive Data Types Table – Default Value, Size, and Range**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** | **Range** |
| **byte** | 0 | 1 byte or 8 bits | -128 to 127 |
| **short** | 0 | 2 bytes or 16 bits | -32,768 to 32,767 |
| **int** | 0 | 4 bytes or 32 bits | 2,147,483,648 to 2,147,483,647 |
| **long** | 0 | 8 bytes or 64 bits | 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| **float** | 0.0f | 4 bytes or 32 bits | 1.4e-045 to 3.4e+038 |
| **double** | 0.0d | 8 bytes or 64 bits | 4.9e-324 to 1.8e+308 |
| **char** | ‘\u0000’ | 2 bytes or 16 bits | 0 to 65536 |
| **boolean** | FALSE | 1 byte or 2 bytes | 0 or 1 |

Wrapper Classes

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

**Need of Wrapper Classes**

There are certain needs for using the Wrapper class in Java as mentioned below:

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Casting - Autoboxing and Unboxing

**1. Autoboxing**

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

// Java program to demonstrate Autoboxing

import java.util.ArrayList;

class Autoboxing {

public static void main(String[] args)

{

char ch = 'a';

// Autoboxing- primitive to Character object

// conversion

Character a = ch;

ArrayList<Integer> arrayList

= new ArrayList<Integer>();

// Autoboxing because ArrayList stores only objects

arrayList.add(25);

// printing the values from object

System.out.println(arrayList.get(0));

}

}

**2. Unboxing**

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

**Example:**

Java

*// Java program to demonstrate Unboxing*

**import** **java.util.ArrayList**;

**class** **Unboxing** {

**public** **static** void main(String[] args)

{

Character ch = 'a';

*// unboxing - Character object to primitive*

*// conversion*

char a = ch;

ArrayList<Integer> arrayList

= **new** ArrayList<Integer>();

arrayList.add(24);

*// unboxing because get method returns an Integer*

*// object*

int num = arrayList.get(0);

*// printing the values from primitive data types*

System.out.println(num);

}

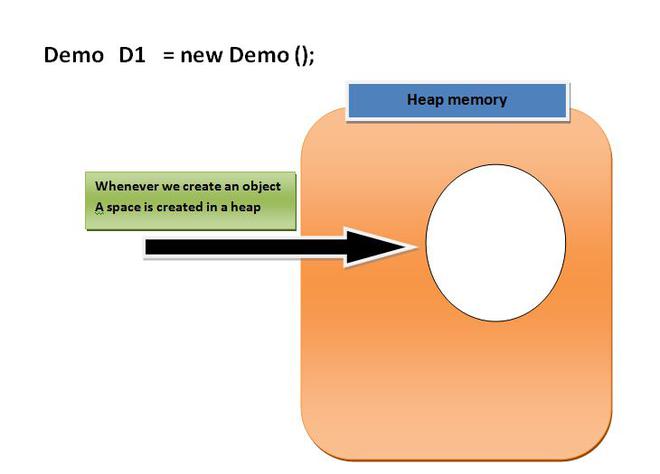
}

More detail refer : [Wrapper Classes in Java - GeeksforGeeks](https://www.geeksforgeeks.org/wrapper-classes-java/)

Reference Variables

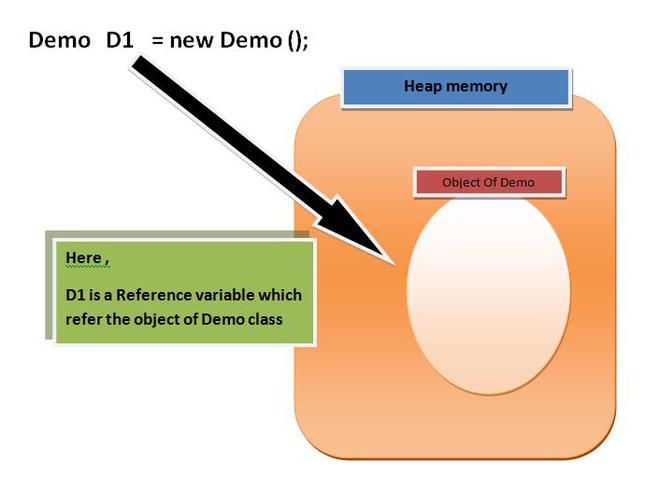
**1.**When we create an object(instance) of class then space is reserved in heap memory. Let’s understand with the help of an example.

*Demo D1 = new Demo();*



Now, The space in the heap Memory is created but the question is **how to access that space?**.

Then, We create a Pointing element or simply called **Reference variable** which simply points out the Object(the created space in a Heap Memory).



**Understanding Reference variable**

**1.** Reference variable is used to point object/values.

**2.**Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java. Reference variables hold the objects/values of reference types in Java.

**3.** Reference variable can also store **null** value. By default, if no object is passed to a reference variable then it will store a null value.

**4.**You can access object members using a reference variable using **dot**syntax.

*<reference variable name >.<instance  variable\_name / method\_name>*

**Example:**

* Java

|  |
| --- |
| // Java program to demonstrate reference  // variable in java    **import** java.io.\*;    **class** Demo {  **int** x = 10;  **int** display()      {          System.out.println("x = " + x);  **return** 0;      }  }    **class** Main {  **public** **static** **void** main(String[] args)      {          Demo D1 = **new** Demo(); // point 1            System.out.println(D1); // point 2            System.out.println(D1.display()); // point 3      }  } |

**Output**

Demo@214c265e

x = 10

0

**Let us see what is actually happening step by step.**

**1.** When we create an object of demo class **new DEMO();**, the default constructor is called and returns a reference of the object, and simply this reference will be stored to the reference variable **D1** (As we know that associativity is Right-hand side to left-hand side).

**2.** The value of a reference variable is a reference. When we attempt to print the value of a reference variable, the output contains the name of the class which has been instantiated concatenated by @ and the hash code created for it by Java: the string **Demo@214c265e** tells us that the given variable is of type Name and its hexadecimal format of hash code is 214c265e.

**3.**At this point we will access the methods **display()** of the class demo using our custom reference variable that we created.

***BINDING UP****: The constructor call returns a value that is a reference to the newly-created object. The equality sign tells the program that the value of the right-hand side expression is to be copied as the value of the variable on the left-hand side. The reference to the newly-created object, returned by the constructor call, is copied as the value of the variable.*

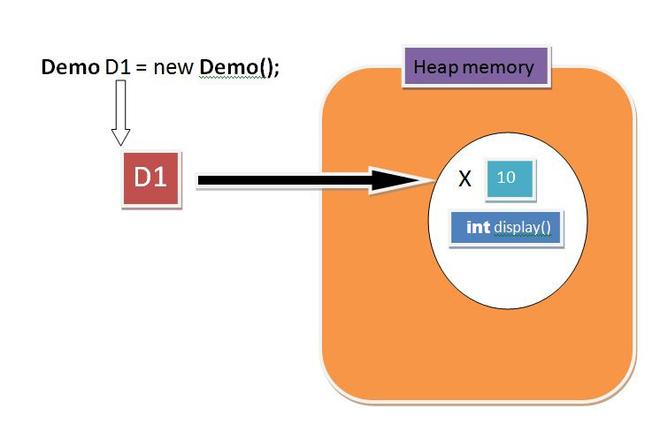
* Java

|  |
| --- |
| **import** java.io.\*;  **class** Demo {  **int** x = 10;    **int** display()      {          System.out.println("x = " + x);  **return** 0;      }  }    **class** Main {  **public** **static** **void** main(String[] args)      {            // create instance          Demo D1 = **new** Demo();            // accessing instance(object) variable          System.out.println(D1.x);            // point 3          // accessing instance(object) method          D1.display();      }  } |

**Output**

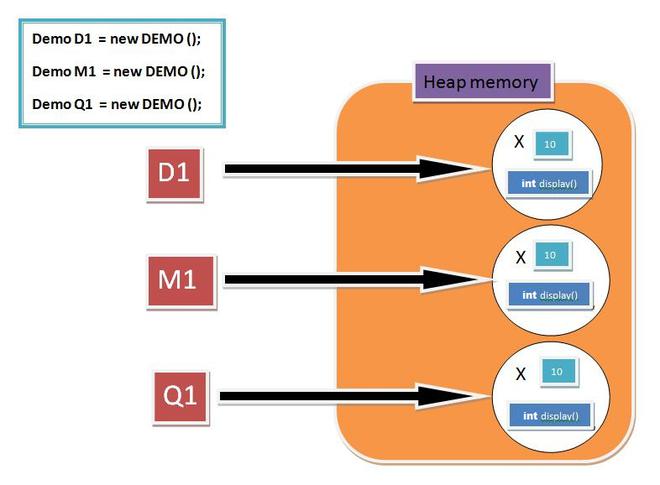
10

x = 10



* Java

|  |
| --- |
| // Accessing instance methods    **import** java.io.\*;  **class** Demo {    **int** x = 10;  **int** display()      {          System.out.println("x = " + x);  **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {            // create instances          Demo D1 = **new** Demo();            Demo M1 = **new** Demo();            Demo Q1 = **new** Demo();      }  } |



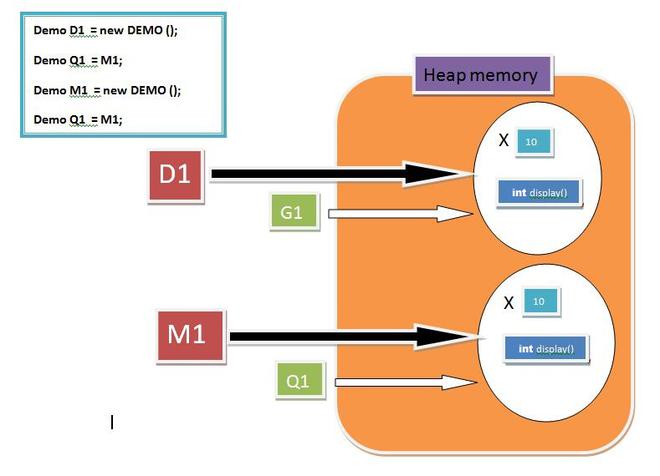
* Java

|  |
| --- |
| // Pointing to same instance memory    **import** java.io.\*;  **class** Demo {  **int** x = 10;  **int** display()      {          System.out.println("x = " + x);  **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {          // create instance          Demo D1 = **new** Demo();            // point to same reference          Demo G1 = D1;            Demo M1 = **new** Demo();            Demo Q1 = M1;            // updating the value of x using G!          // reference variable          G1.x = 25;            System.out.println(G1.x); // Point 1            System.out.println(D1.x); // Point 2      }  } |

**Output**

25

25



***Note:***

*Here we pass G1 and Q1 reference variable point out the same object respectively. Secondly****At Point 1****we try to get the value of the object with G1 reference variable which shows it as****25****and****At Point 2****we try to get the value of an object with D1 reference variable which shows it as****25****as well. This will prove that the modification in the object can be done by using any reference variable but the condition is it should hold the same reference.*

**More on Reference Variable**

**1. Reference Variable as Method Parameters:**

As the value of a primitive variable is directly stored in the variable, whereas the value of a reference variable holds a reference to an object. We also mentioned that assigning a value with the equality sign copies the value (possibly of some variable) on the right-hand side and stores it as the value of the left-hand-side variable. A similar kind of copying occurs during a method call. Regardless of whether the variable is primitive or reference type, a copy of the value is passed to the method’s argument and copied to that argument.

***Note:****Java only supports pass by value.*

But we know that the reference variable holds the reference of an instance(OBJECT) so a copy of the reference is passed to the method’s argument.

**Example:**

* Java

|  |
| --- |
| // Pass by reference and value    **import** java.io.\*;  **class** Demo {  **int** x = 10;  **int** y = 20;    **int** display(Demo A, Demo B)      {          //  Updating value using argument          A.x = 95;            System.out.println("x = " + x);            System.out.println("y = " + y);    **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {          Demo C = **new** Demo();            Demo D = **new** Demo();            // updating value using primary reference          // variable          D.y = 55;            C.display(C, D); // POINT 1            D.display(C, D); // POINT 2      }  } |

**Output**

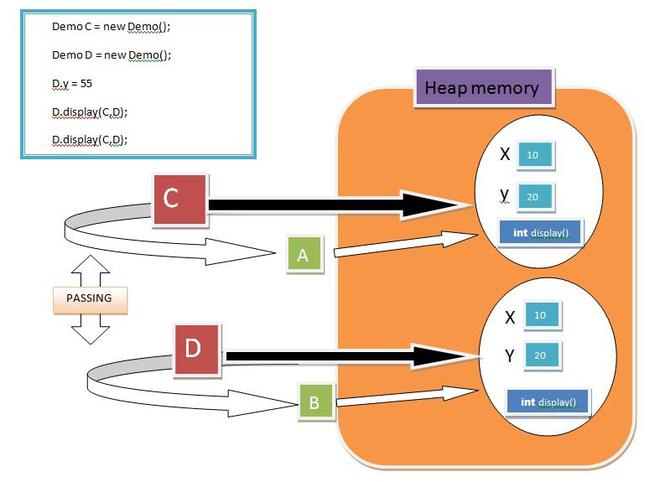
x = 95

y = 20

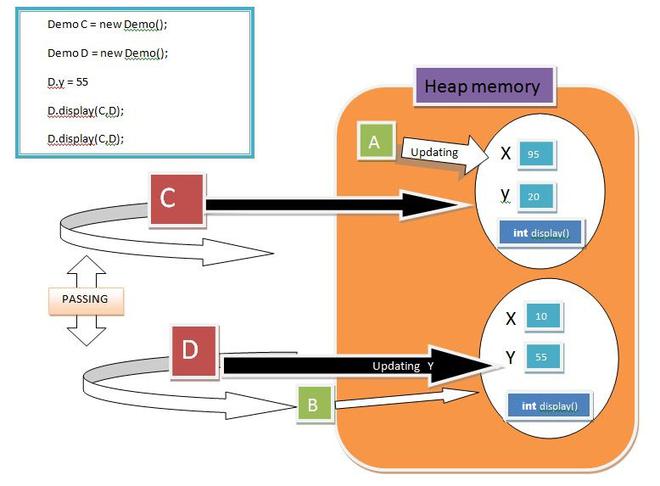
x = 10

y = 55

**SCENE 1 :**



**SCENE 2:**



Now, What is going on here, when we pass the reference to the method it will copy to the reference variable defined in the method signature and After that, they also have access to the object members. Here, We defined two instances named **C and D**. Afterwards we pass C and D to the method which further gives reference to **A and B**

**At Point 1:**A will update the value of x from 10 to 95, hence C.display() will show 95 20 but in another object D we update the value of x through D only from y =20 to 55, hence D, display() will show 10 and 55.

***Note:****Any Object Updation will not affect the other object’s member.*

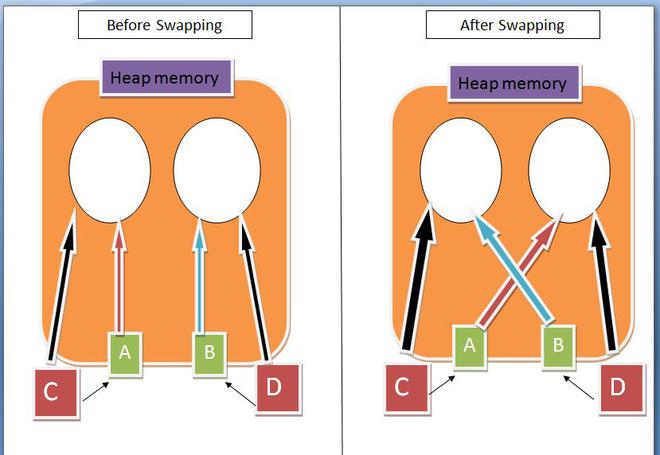
**2. What if we swap the reference variables with the help of the Swap Method?**

The fact is if we try to swap the reference variable, then they just swap their Pointing element there is no effect on the address of reference variable and object(Instance) Space. Let’s Understand It with the help of an example:

* Java

|  |
| --- |
| // Swapping object references    **import** java.io.\*;  **class** Demo {        // Swapping Method  **int** Swap(Demo A, Demo B)      {          Demo temp = A;          A = B;          B = temp;  **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {          Demo C = **new** Demo();            Demo D = **new** Demo();            // Passing C and reference variables          // to Swap method          C.Swap(C, D);      }  } |

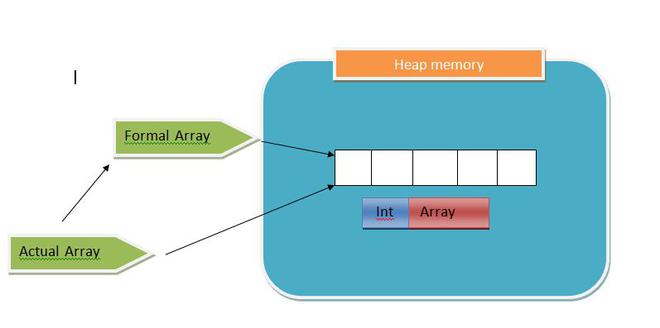
*Here we created, two instances of demo class and passes it to swap method, further those C and D will copy their references to A and B respectively.Before swapping A point to C’s(Object) and B point to D’s(Object). After we perform swapping on A and B, A will now point D’s(Object) and B will Point C’s Object. As described in the figure.*



***Note:****There is no swapping between Variables, They only change their References.*

**3. What if we pass arrays to the method will it be able to update the Actual Array’s values, even we know that a copy of the array is a pass to the formal Array?**

The answer is **YES,**the values will be updated by Formal parameter, The Fact is, When we create an Array, a memory is assigned to the array of the desired size, and it returns the reference of the first array’s element that is the base address that will store to the Formal Array(Method argument). As we learned earlier every pointing reference variable can change or update the object.



**Example:**

* Java

|  |
| --- |
| **import** java.io.\*;  **class** Demo {  **int** arrayUpdate(**int**[] formalArray)      {          formalArray[2] = 99;          formalArray[4] = 77;  **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {          Demo d1 = **new** Demo();  **int**[] actualArray = { 1, 2, 3, 4, 5 };    **for** (**int** items : actualArray)              System.out.print(items                               + " , "); // printing array            System.out.println();          d1.arrayUpdate(actualArray);          System.out.println();    **for** (**int** items : actualArray)              System.out.print(items                               + " , "); // printing array      }  } |

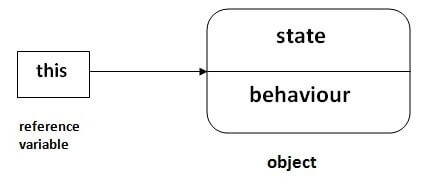
**Output**

1 , 2 , 3 , 4 , 5 ,

1 , 2 , 99 , 4 , 77 ,

**4.** **this and super keywords are also Pointing Elements.**

[this](https://www.geeksforgeeks.org/this-reference-in-java/) keyword. In java, **this** is a reference variable that refers to the current object.



[super](https://www.geeksforgeeks.org/super-keyword/) is used to refer immediate parent class instance variable. We can use the super keyword to access the data member or field of the parent class. It is used if parent class and child class have the same fields.



**5.**[**null**](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/interesting-facts-about-null-in-java/&sa=U&ved=2ahUKEwjmxOyn9-jsAhUhyjgGHbTdAo0QFjAAegQIBRAC&usg=AOvVaw28o5zUru7d2Ax5IBZ6WolM)**value of a reference variable.**

***Demo obj = null ;***

**A.** The **null** reference can be set as the value of any reference type variable.

**B.** The object whose name is **obj** is referred to by nobody. In other words, the object has become **garbage**. In the Java programming language, the programmer need not worry about the program’s memory use. From time to time, the automatic garbage collector of the Java language cleans up the objects that have become garbage. If the garbage collection did not happen, the garbage objects would reserve a memory location until the end of the program execution.

* Java

|  |
| --- |
| // null in java    **import** java.io.\*;  **class** Demo {  **int** x = 10;  **int** display()      {          System.out.println("x = " + x);  **return** 0;      }  }  **class** Main {  **public** **static** **void** main(String[] args)      {          Demo obj = **null**;            // accessing instance(object) method          Kuchbhi.display();      }  } |

**Output**

Exception in thread "main" java.lang.NullPointerException

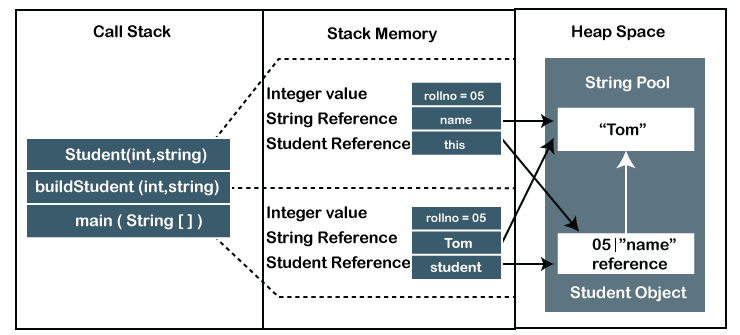
at Main.main(File.java:17)

Java Result: 1

Here, we try to access objects’ members by a reference variable that is pointing nothing(null), and hence it shows **NullPointerException.**Now if you get the error, the first step is to look for variables whose value could be null. Fortunately, the error message is useful: it tells which row caused the error. Just try it out yourself!

[Reference Variable in Java - GeeksforGeeks](https://www.geeksforgeeks.org/reference-variable-in-java/)

Value and Reference Types (Stack/Heap)



Difference Between Stack and Heap Memory

The following table summarizes all the major differences between stack memory and heap space.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Stack Memory** | **Heap Space** |
| **Application** | It stores items that have a very short life such as **methods, variables,** and **reference variables** of the objects. | It stores **objects** and Java Runtime Environment (**JRE**) classes. |
| **Ordering** | It follows the **LIFO** order. | It does not follow any order because it is a dynamic memory allocation and does not have any fixed pattern for allocation and deallocation of memory blocks. |
| **Flexibility** | It is **not flexible** because we cannot alter the allocated memory. | It is **flexible** because we can alter the allocated memory. |
| **Efficiency** | It has **faster** access, allocation, and deallocation. | It has **slower** access, allocation, and deallocation. |
| **Memory Size** | It is **smaller** in size. | It is **larger** in size. |
| **Java Options Used** | We can increase the stack size by using the JVM option -Xss. | We can increase or decrease the heap memory size by using the -[Xmx](https://www.javatpoint.com/java-xmx) and -Xms JVM options. |
| **Visibility or Scope** | The variables are visible only to the owner thread. | It is visible to all threads. |
| **Generation of Space** | When a thread is created, the operating system automatically allocates the stack. | To create the heap space for the application, the language first calls the operating system at run time. |
| **Distribution** | Separate stack is created for each object. | It is shared among all the threads. |
| **Exception Throws** | JVM throws the **java.lang.StackOverFlowError** if the stack size is greater than the limit. To avoid this error, increase the stack size. | JVM throws the **java.lang.OutOfMemoryError** if the JVM is unable to create a new native method. |
| **Allocation/ Deallocation** | It is done automatically by the **compiler**. | It is done manually by the **programmer**. |
| **Cost** | Its cost is **less**. | Its cost is **more** in comparison to stack. |
| **Implementation** | Its implementation is **hard**. | Its implementation is **easy**. |
| **Order of allocation** | Memory allocation is **continuous**. | Memory allocated in **random** order. |
| **Thread-Safety** | It is thread-safe because each thread has its own stack. | It is not thread-safe, so properly synchronization of code is required. |

Strings and the String pool

`**Strings in Java**

* **Strings** are objects that represent sequences of characters.
* They are immutable, meaning once a String object is created, it cannot be changed.

**String Pool**

* The **String Pool** (or intern pool) is a special memory region where Java stores string literals.
* When you create a string literal, Java checks the pool to see if an identical string already exists. If it does, Java returns a reference to the existing string. If not, it creates a new string in the pool.

String str1 = "Hello";

String str2 = "Hello";

// Both str1 and str2 point to the same object in the string pool

System.out.println(str1 == str2); // true

String str3 = new String("Hello");

// str3 points to a new object, not the one in the string pool

System.out.println(str1 == str3); // false

**Benefits**

* **Memory Efficiency**: Reusing strings from the pool saves memory.
* **Performance**: Comparing strings using == is faster when they are from the pool.

StringBuilder & StringBuffer

**StringBuilder**

* **StringBuilder** is a mutable sequence of characters.
* It is not synchronized, which means it is not thread-safe.
* It is faster than StringBuffer because of the lack of synchronization.

**StringBuffer**

* **StringBuffer** is also a mutable sequence of characters.
* It is synchronized, meaning it is thread-safe.
* It is slower than StringBuilder due to the overhead of synchronization.

**Key Differences**

1. **Thread Safety**:
   * StringBuilder is not thread-safe.
   * StringBuffer is thread-safe.
2. **Performance**:
   * StringBuilder is faster.
   * StringBuffer is slower due to synchronization.

// Using StringBuilder

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb.toString()); // Outputs: Hello World

// Using StringBuffer

StringBuffer sbf = new StringBuffer("Hello");

sbf.append(" World");

System.out.println(sbf.toString()); // Outputs: Hello World

**When to Use**

* Use **StringBuilder** when you do not need thread safety.
* Use **StringBuffer** when you need thread safety.

Control Flow - Conditional Statements (if/else if/else, switch/case)

**if/else if/else**

* The if statement evaluates a condition and executes a block of code if the condition is true.
* The else if statement allows you to check multiple conditions.
* The else statement executes a block of code if none of the previous conditions are true.

**switch/case**

* The switch statement allows you to select one of many code blocks to be executed.
* It works with byte, short, char, int, and enumerated types, as well as the String class and a few special classes.

int day = 3;

String dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

break;

}

System.out.println(dayName); // Outputs: Wednesday

**Key Points**

* Use if/else if/else for conditions that are not mutually exclusive or when you have complex conditions.
* Use switch/case for simpler, mutually exclusive conditions, especially when dealing with a single variable that can take on a limited set of values.

Logical Operators (&&, ||, !&, !|)

**Logical AND (&&)**

* The && operator returns true if both operands are true.
* If either operand is false, it returns false.

**Example**

**Java**

boolean a = true;

boolean b = false;

System.out.println(a && b); // Outputs: false

System.out.println(a && true); // Outputs: true

**Logical OR (||)**

* The || operator returns true if at least one of the operands is true.
* If both operands are false, it returns false.

**Example**

**Java**

boolean a = true;

boolean b = false;

System.out.println(a || b); // Outputs: true

System.out.println(b || false); // Outputs: false

**Logical NOT (!)**

* The ! operator inverts the value of a boolean expression.
* If the expression is true, it returns false, and vice versa.

**Example**

**Java**

boolean a = true;

System.out.println(!a); // Outputs: false

System.out.println(!false); // Outputs: true

**Common Mistakes**

* !&**and**!|: These are not valid logical operators in Java. The correct operators are &&, ||, and !.

**Combining Logical Operators**

You can combine these operators to form complex logical expressions.

**Example**

**Java**

boolean a = true;

boolean b = false;

boolean c = true;

System.out.println((a && b) || c); // Outputs: true

System.out.println(!(a && b) || c); // Outputs: true

Mathematical Operators (+, -, \*, /, %, =)

**Addition (+)**

* Adds two operands.

**Java**

int a = 5;

int b = 3;

int sum = a + b; // sum is 8

**Subtraction (-)**

* Subtracts the second operand from the first.

**Java**

int a = 5;

int b = 3;

int difference = a - b; // difference is 2

**Multiplication (\*)**

* Multiplies two operands.

**Java**

int a = 5;

int b = 3;

int product = a \* b; // product is 15

**Division (/)**

* Divides the first operand by the second.
* Be cautious of dividing by zero, which will throw an ArithmeticException.

**Java**

int a = 6;

int b = 3;

int quotient = a / b; // quotient is 2

**Modulus (%)**

* Returns the remainder of the division of the first operand by the second.

**Java**

int a = 5;

int b = 3;

int remainder = a % b; // remainder is 2

**Assignment (=)**

* Assigns the value on the right to the variable on the left.

**Java**

int a = 5; // a is assigned the value 5

Comparison Operators (==, <, >, >=, <=, !=)

**Equal to (==)**

* Checks if two values are equal.

**Java**

int a = 5;

int b = 5;

System.out.println(a == b); // Outputs: true

**Not equal to (!=)**

* Checks if two values are not equal.

**Java**

int a = 5;

int b = 3;

System.out.println(a != b); // Outputs: true

**Greater than (>)**

* Checks if the value on the left is greater than the value on the right.

**Java**

int a = 5;

int b = 3;

System.out.println(a > b); // Outputs: true

**Less than (<)**

* Checks if the value on the left is less than the value on the right.

**Java**

int a = 5;

int b = 3;

System.out.println(a < b); // Outputs: false

**Greater than or equal to (>=)**

* Checks if the value on the left is greater than or equal to the value on the right.

**Java**

int a = 5;

int b = 5;

System.out.println(a >= b); // Outputs: true

**Less than or equal to (<=)**

* Checks if the value on the left is less than or equal to the value on the right.

**Java**

int a = 5;

int b = 5;

System.out.println(a <= b); // Outputs: true

**Example Combining Comparison Operators**

You can use these operators in conditional statements to control the flow of your program.

**Java**

int a = 10;

int b = 5;

if (a > b) {

System.out.println("a is greater than b");

} else if (a < b) {

System.out.println("a is less than b");

} else {

System.out.println("a is equal to b");

}