**Introduction:**

**Java Language History?**Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle) and released in **1995** as a core component of Sun Microsystems' Java platform.

It started with version 1.0 and has a latest version of 16.

**How a Java Program runs?**

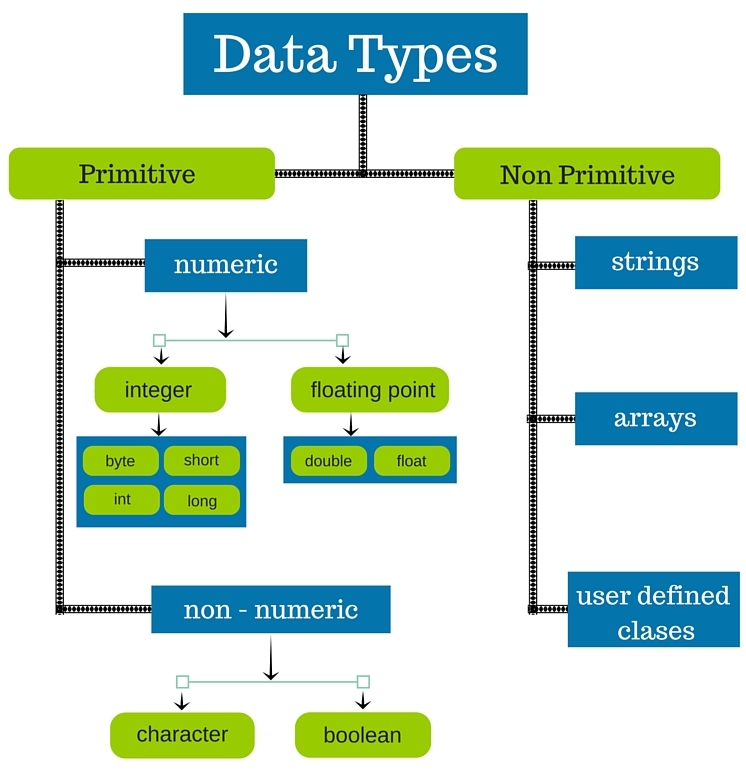
**JDK, JRE, JVM**Diagram

Description automatically generated

**Data Types and Variables in Java?**

Data type specifies the size and type of values that can be stored in an identifier.

Different types of Data Types in Java are described as below:



### **byte**

* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
* Example: byte a = 100, byte b = -50

### **short**

* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer.
* Default value is 0.
* Example: short s = 10000, short r = -20000

### **int**

* Minimum value is - 2,147,483,648 (-2^31)
* Maximum value is 2,147,483,647(inclusive) (2^31 -1)
* Integer is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0
* Example: int a = 100000, int b = -200000

### **long**

* Minimum value is -9,223,372,036,854,775,808(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
* This type is used when a wider range than int is needed
* Default value is 0L
* Example: long a = 100000L, long b = -200000L

### **float**

* Float is mainly used to save memory in large arrays of floating point numbers
* Range: 3.40282347 x 1038, 1.40239846 x 10-45
* Default value is 0.0f
* Example: float f1 = 234.5f

### **double**

* This data type is generally used as the default data type for decimal values, generally the default choice
* Range: 1.7976931348623157 x 10308, 4.9406564584124654 x 10-324
* Default value is 0.0d
* Example: double d1 = 123.4

### **boolean**

* boolean data type represents one bit of information
* There are only two possible values: true and false
* This data type is used for simple flags that track true/false conditions
* Default value is false
* Example: boolean one = true

### **char**

* Char data type is used to store any character
* Example: char letterA = 'A'

**What are Java Modifiers and there access properties?**

Access Modifiers in Java. Access modifiers are keywords in Java that are used to set accessibility. An access modifier restricts the access of a class, constructor, data member and method in another class. Java language has four access modifier to control access level for classes and its members.

We have Access Modifiers which controls the access level as discussed above and below is the pictorial representation for the same.

Table

Description automatically generated

We also have Non Access Modifiers which are listed below:

* ABSTRACT
* FINAL
* STATIC

These concepts can be covered ahead.

**What are Java Operators?**

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators

Arithmetic Operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** | **Try it** |
| + | Addition | Adds together two values | x + y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_add) |
| - | Subtraction | Subtracts one value from another | x - y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_sub) |
| \* | Multiplication | Multiplies two values | x \* y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_mult) |
| / | Division | Divides one value by another | x / y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_div) |
| % | Modulus | Returns the division remainder | x % y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_mod) |
| ++ | Increment | Increases the value of a variable by 1 | ++x | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_inc) |
| -- | Decrement | Decreases the value of a variable by 1 | --x |  |

Java Assignment Operators

Assignment operators are used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** | **Try it** |
| = | x = 5 | x = 5 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass1) |
| += | x += 3 | x = x + 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass2) |
| -= | x -= 3 | x = x - 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass3) |
| \*= | x \*= 3 | x = x \* 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass4) |
| /= | x /= 3 | x = x / 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass5) |
| %= | x %= 3 | x = x % 3 |  |

Java Comparison Operators

Comparison operators are used to compare two values:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| == | Equal to | x == y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare1) |
| != | Not equal | x != y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare2) |
| > | Greater than | x > y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare3) |
| < | Less than | x < y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare4) |
| >= | Greater than or equal to | x >= y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare5) |
| <= | Less than or equal to | x <= y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare6) |

Java Logical Operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** | **Try it** |
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_logical1) |
| || | Logical or | Returns true if one of the statements is true | x < 5 || x < 4 |  |

Logical operators are used to determine the logic between variables or values:

**What are Java Keywords?**

Java keywords are also known as reserved words. Keywords are particular words which acts as a key to a code. These are predefined words by Java so it cannot be used as a variable or object name.

Examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new |  |
| assert | default |  | package |  |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
|  |  |  | return |  |
| catch | extends | int | short | try |
| char | final | interface | static | void |
|  |  |  |  |  |
| class | finally | long |  |  |
|  | float |  | super | while |

**OOPS Concepts?**Java is based on OOPS Concepts and it is a very important concept. OOPS concept is generally divided as below:

* Inheritance
* Polymorphism (Method Overloading & Method Overriding)
* Abstraction
* Encapsulation

Inheritance is one of the Basic Concepts of OOPs in which one object acquires the properties and behaviors of the parent object. It’s creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

**INHERITANCE:**

**Inheritance** is a mechanism in which **one class acquires the property of another class**. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates **Reusability and is an important concept of OOPs.**

Keyword used for Inheritance : **extends**

**Class B extends Class A(){**

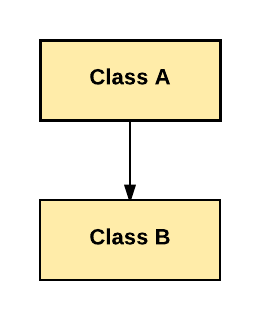
**}**

## Types of Inheritance

There are Various types of inheritance in Java:

### **Single Inheritance**:

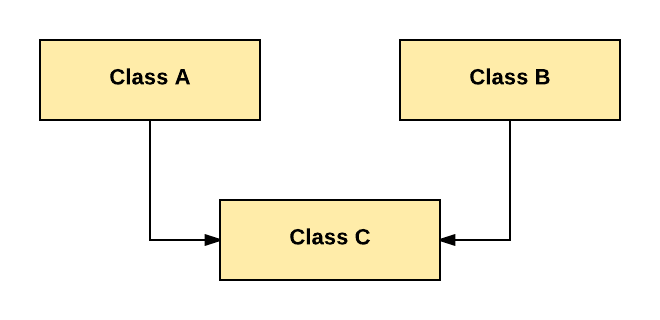
In Single Inheritance one class extends another class (one class only).

[](https://www.guru99.com/images/java/single_inheritance.png)Single Inheritance

In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.nt, Install Software and Email - Linux Tutorial 7

### **Multiple Inheritance: //Interface**

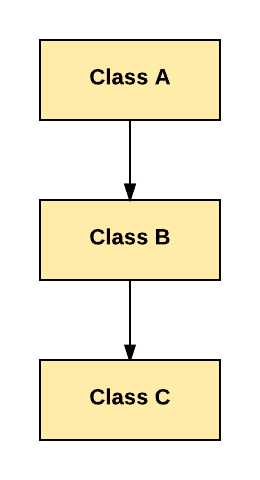
In Multiple Inheritance, one class extending more than one class. **Java does not support multiple inheritance.**

[](https://www.guru99.com/images/java/multiple.png)Multiple Inheritance

As per above diagram, Class C extends Class A and Class B both.

### **Multilevel Inheritance**:

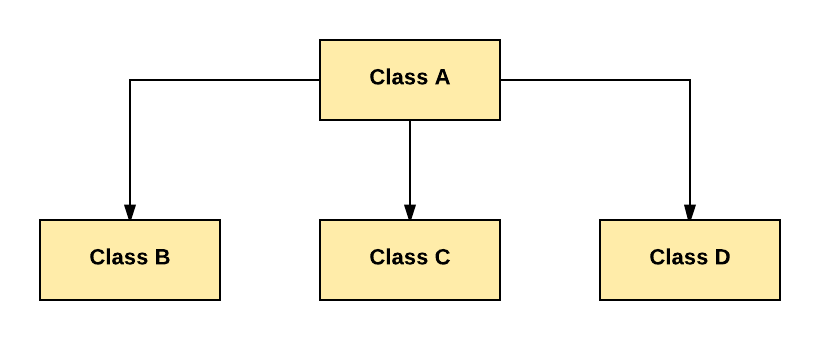
In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

[](https://www.guru99.com/images/java/multilevel.png)

As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

### **Hierarchical Inheritance:**

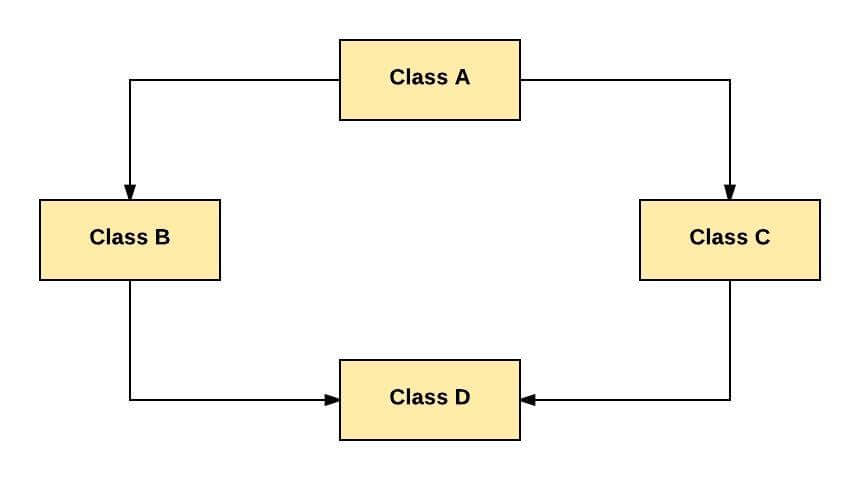
In Hierarchical Inheritance, one class is inherited by many sub classes.

[](https://www.guru99.com/images/java/hierarchy.png)**Hierarchical Inheritance**

As per above example, Class B, C, and D inherit the same class A.

### **Hybrid Inheritance:**

Hybrid inheritance is a combination of Single and **Multiple inheritance**.

[](https://www.guru99.com/images/java/hybrid.jpeg)Hybrid Inheritance

As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

**Note:** **Java doesn't support hybrid/Multiple inheritance.**

**Polymorphism (Method Overloading & Method Overriding)**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

**Method Overloading:**   
With**method overloading**, multiple methods can have the same name with different parameters:

Valid Conditions for a method to be overloaded:

1. Different **number of Arguments** in the method

class MethodOverloading {

private static void display(int a){

System.out.println("Arguments: " +a);

}

private static void display(int a, int b){

System.out.println("Arguments: " + a + " and " + b);

}

public static void main(String[] args) {

display(1);

display(1, 4);

}

}

2. **Different type of Arguments** in the method

class MethodOverloading {

// this method accepts int

private static void display(int a){

System.out.println("Got Integer data.");

}

// this method accepts String object

private static void display(String a){

System.out.println("Got String object.");

}

public static void main(String[] args) {

display(1);

display("Hello");

}

}

3. Different **sequence of arguments** in the method

class MethodOverloading {

// this method accepts int

private static void display(int a, String b){

System.out.println("This is the first display method called");

}

// this method accepts String object

private static void display(String a, int b){

System.out.println("This is the second display method called ");

}

public static void main(String[] args) {

display(1,’Hello’);

display("Hello",2);

}

}

**Method Overriding**: Overriding is a feature that **allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes**.

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

|  |
| --- |
| // A Simple Java program to demonstrate  // method overriding in java    // Base Class  class Parent {      void show()      {          System.out.println("Parent's show()");      }  }    // Inherited class  class Child **extends** Parent {      // This method overrides show() of Parent      @Override      void show()      {          System.out.println("Child's show()");      }  }    // Driver class  class Main {      public static void main(String[] args)      {          // If a Parent type reference refers          // to a Parent object, then Parent's          // show is called          Parent obj1 = new Parent();              // If a Parent type reference refers          // to a Child object Child's show()          // is called. This is called RUN TIME          // POLYMORPHISM.          Parent obj2 = new Child();          obj2.show();      }  }  ChildClass child\_c= new ChildClass(); ParentClass child\_c= new ParentClass(); ParentClass pc=new ChildClass(); |

**Output:**

Parent's show()

Child's show()

**Abstraction Concept?  
abstraction** is a process of hiding the implementation details from the user, only the functionality will be provided to the user.

* **When the class has both the methods i.e. : implemented/non implemented** then we can say that we are having abstraction where the unimplemented method is abstract method & class is also called Abstraction class.
* There is **partial abstraction** achieved in abstraction as compared to **100% achieved in Interface**.
* abstract keyword is used with method/class.
* Use inheritance concept to extend the abstract class.
* You cannot create object for a class which is marked as abstract class.
* private is not allowed to be used as access modifier for a method (public/protected is allowed- Not sure of the default type)
* same for variables as above as well

abstract class Animal {

public abstract void animalSound();

public **void** sleep() {

System.out.println("Zzz");

}

}

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // will generate an error

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Sleep is good for health");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig is a PIG");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

Output:

The pig is a PIG  
Sleep is good for health

**Encapsulation Concept?**

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class.

* declare class variables/attributes as private
* provide public get and set methods to access and update the value of a private variable

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.setName("Ajay"); // Set the value of the name variable to "Nitish"

System.out.println(myObj.getName());

}

}

// Outputs "Ajay"

**Interface Concept?**

An **Interface in Java** programming is defined as an abstract type used to specify the behaviour of a class. A Java interface contains static constants and abstract methods. **A class can implement multiple interfaces**. In Java, interfaces are declared using the interface keyword. All methods in the interface are implicitly public and abstract.

* Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class.
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public.
* Interface attributes are by default public, static and final.
* An interface cannot contain a constructor (as it cannot be used to create objects)

// Interface

interface Animal {

public void animalSound(); // interface method (does not have a body)

public void sleep(); // interface method (does not have a body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("Animal is making a sound");

}

public void sleep() {

// The body of sleep() is provided here

System.out.println("Animal is sleeping");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

Output:

Animal is making a sound  
Animal is sleeping.

**Local and Global Variables?**

There are 3 types of variables in Java:

1. Local Variable  
2. Static Variable  
3. Instance Variable

**LOCAL VARIABLE**

**Local Variable** is defined as a type of variable declared within programming block or subroutines. It can only be used inside the subroutine or code block in which it is declared. The local variable exists until the block of the function is under execution. After that, it will be destroyed automatically.

**Example of Local Variable**

public int add(){

int a =4;

int b=5;

return a+b;

}

Here, 'a' and 'b' are local variables

**GLOBAL VARIABLE**

A **Global Variable** in the program is a variable defined outside the subroutine or function. It has a global scope means it holds its value throughout the lifetime of the program. Hence, it can be accessed throughout the program by any function defined within the program, unless it is shadowed.

Example:

int a =4;

int b=5;

public int add(){

return a+b;

}

Here, 'a' and 'b' are global variables.

**STATIC VARIABLE**

Static Variable is variable which belongs to the class and initialized only once at the start of the execution. It is a variable which belongs to the class and not to object (instance). Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.

* A single copy to be shared by all instances of the class.
* A static variable can be accessed directly by the class name and does not need any object.
* class JavaExample{
* static int age;
* static String name;
* //This is a Static Method
* static void disp(){
* System.out.println("Age is: "+age);
* System.out.println("Name is: "+name);
* }
* // This is also a static method
* public static void main(String args[])
* {
* age = 30;
* name = "Steve";
* disp();
* }
* }

Output:

* Age is: 30
* Name is: Steve

**STATIC METHODS**

A static method belongs to the class rather than the object of a class. A static method **can be invoked without the need for creating an instance of a class**. A static method can access static data member and can change the value of it.

**NON STATIC METHODS**

A non-static method does not have the keyword static before the name of the method. A non-static method belongs to an object of the class and you have to create an instance of the class to access it. Non-static methods can access any static method and any static variable without creating an instance of the class.

public class JavaTester {

   public static void main(String args[]) {

      Tiger.roar();

      Tiger tiger = new Tiger();

      tiger.eat();

   }

}

class Tiger {

   public void eat(){

      System.out.println("Tiger eats");

   }

   public static void roar(){

      System.out.println("Tiger roars");

   }

}

**DIFFERENCE BETWEEN STATIC AND NON STATIC METHODS**

| **Sr. No.** | **Key** | **Static** | **Non-Static** |
| --- | --- | --- | --- |
| 1 | Access | A static method can access only static members and can not access non-static members. | A non-static method can access both static as well as non-static members. |
| 2 | Binding | Static method uses complie time binding or early binding. | Non-static method uses run time binding or dynamic binding. |
| 3 | Overriding | A static method cannot be overridden being compile time binding. | A non-static method can be overridden being dynamic binding. |
| 4 | Memory allocation | Static method occupies less space and memory allocation happens once. | A non-static method may occupy more space. Memory allocation happens when method is invoked and memory is deallocated once method is executed completely. |
| 5 | Keyword | A static method is declared using static keyword. | A normal method is not required to have any special keyword. |

**LOOPS**

**If**

Use the if statement to specify a block of Java code to be executed if a condition is true.

int x = 20;

int y = 18;

if (x > y) {

System.out.println("x is greater than y");

}

**Else**

Use the else statement to specify a block of code to be executed if the condition is false.

int time = 20;

if (time < 18) {

System.out.println("Good day.");

} else {

System.out.println("Good evening.");

}

// Outputs "Good evening."

**Else if**

Use the else if statement to specify a new condition if the first condition is false.

int time = 22;

if (time < 10) {

System.out.println("Good morning.");

} else if (time < 20) {

System.out.println("Good day.");

}

else {

System.out.println("Good evening.");

}

// Outputs "Good evening."

**While**

The while loop loops through a block of code as long as a specified condition is true:

int i = 0;

while (i <= 5) {

System.out.println(i);

i++;

}

**For loop**

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

for (int i = 0; i <= 5; i++) {

System.out.println(i);

}

**Final & Finally**

**Final:**

In the Java programming language, the final keyword is used in several contexts to define an entity that can only be assigned once. Once a final variable has been assigned, it always contains the same value.

public class Main {

**final** int x = 10;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 25; // will generate an error: cannot assign a value to a **final** variable

System.out.println(myObj.x);

}

}

**Finally**:

Java finally block is a block that is used to execute important code such as closing connection, stream etc. **Java finally block is always executed whether exception is handled or not.**

**Execute code, after try...catch, regardless of the result:**

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("The 'try catch' is finished.");

}

**EXCEPTIONS AND EXCEPTION HANDLING**

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

* Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

### Example

[Live Demo](http://tpcg.io/9u4a5O)

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]){

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If you try to compile the above program, you will get the following exceptions.

### Output

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

Note − Since the methods read() and close() of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

* Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

### Example

[Live Demo](http://tpcg.io/7CUnsL)

public class Unchecked\_Demo {

public static void main(String args[]) {

int num[] = {1, 2, 3, 4};

System.out.println(num[5]);

}

}

If you compile and execute the above program, you will get the following exception.

### Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

* Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

### **Example**

import java.io.\*;

public class className {

public void deposit(double amount) throws RemoteException {

// Method implementation

throw new RemoteException();

}

// Remainder of class definition

}

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

**EXAMPLES**

public class JavaTester{

   public void checkAge(int age){

      if(age<18)

         throw new ArithmeticException("Not Eligible for voting");

      else

         System.out.println("Eligible for voting");

   }

   public static void main(String args[]){

      JavaTester obj = new JavaTester();

      obj.checkAge(13);

      System.out.println("End Of Program");

   }

}

## Output

Exception in thread "main" java.lang.ArithmeticException:

Not Eligible for voting

at JavaTester.checkAge(JavaTester.java:4)

at JavaTester.main(JavaTester.java:10)

public class JavaTester{

   public int division(int a, int b) throws ArithmeticException{

      int t = a/b;

      return t;

   }

   public static void main(String args[]){

      JavaTester obj = new JavaTester();

      try{

         System.out.println(obj.division(15,0));

      }

      catch(ArithmeticException e){

         System.out.println("You shouldn't divide number by zero");

      }

   }

}

## Output

You shouldn't divide number by zero

**CONSTRUCTOR**

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

// Create a Main class

public class Main {

int x; // Create a class attribute

// Create a **class constructor** for the Main class

public Main() {

x = 5; // Set the initial value for the class attribute x

}

public static void main(String[] args) {

Main myObj = new Main(); // Create an object of class Main (This will **call the constructor**)

System.out.println(myObj.x); // Print the value of x

}

}

// Outputs 5

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

public class Main {

int x;

public Main(int y) {

x = y;

}

public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println(myObj.x);

}

}

// Outputs 5

**This:**

The this keyword refers to the current object in a method or constructor.

public class Main {

int x;

// Constructor with a parameter

public Main(int x) {

this.x = x;

}

// Call the constructor

public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println("Value of x = " + myObj.x);

}

}

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

**Java Arrays**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

To create an array of integers, you could write:

int[] myNum = {10, 20, 30, 40};

## Access the Elements of an Array

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[3]);

// Outputs Volvo

## Array Length

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4

## Loop Through an Array

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.length; i++) {

System.out.println(cars[i]);

}

## Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

### **Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7},{},{},{} };

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

### **Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

int x = myNumbers[0][2];

System.out.println(x); // Outputs 3

**COLLECTIONS**

The Java collections framework is a set of classes and interfaces that implement commonly reusable collection data structures.

**Diagram

Description automatically generated**

**Differences**

**Graphical user interface, table

Description automatically generated with medium confidence**

**STRINGS & METHODS**

**String str=’Nitish’**

|  |
| --- |
| // Java code to illustrate String  import java.io.\*;  import java.lang.\*;    class Test {      public static void main(String[] args)      {          // Declare String without using new operator          String s = "GeeksforGeeks";            // Prints the String.          System.out.println("String s = " + s);            // Declare String using new operator          String s1 = new String("GeeksforGeeks");            // Prints the String.          System.out.println("String s1 = " + s1);      }  } |

**Output:**

String s = GeeksforGeeks

String s1 = GeeksforGeeks

The String class has a set of you can use on strings.

|  |  |  |
| --- | --- | --- |
| **Method** | built-in methods that **Description** | **Return Type** |
| [charAt()](https://www.w3schools.com/java/ref_string_charat.asp) | Returns the character at the specified index (position) | char |
| [compareTo()](https://www.w3schools.com/java/ref_string_compareto.asp) | Compares two strings | int |
| [compareToIgnoreCase()](https://www.w3schools.com/java/ref_string_comparetoignorecase.asp) | Compares two strings ignoring case differences | int |
| [concat()](https://www.w3schools.com/java/ref_string_concat.asp) | Appends a string to the end of another string | String |
| [contains()](https://www.w3schools.com/java/ref_string_contains.asp) | Checks whether a string contains a sequence of characters | boolean |
| [contentEquals()](https://www.w3schools.com/java/ref_string_contentequals.asp) | Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer | boolean |