# Primitive data types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

int gear = 1;

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

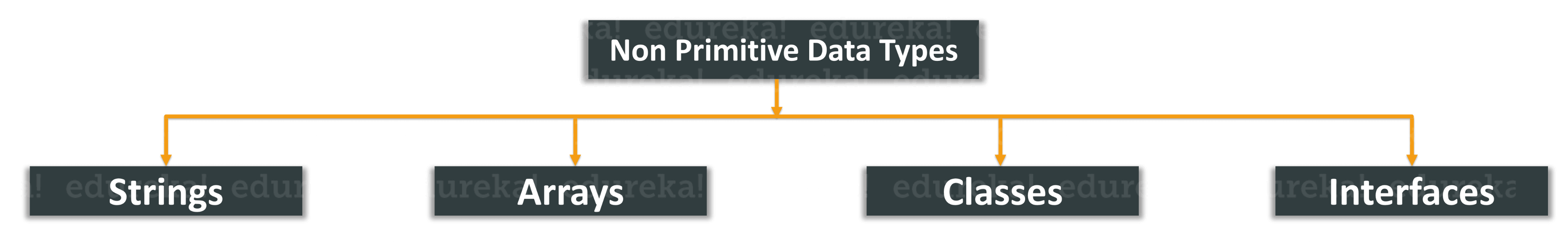
1. **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
2. **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
3. **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.
4. **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. Use this data type when you need a range of values wider than those provided by int. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.
5. **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java

platform.

1. **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
2. **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
3. **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

# Non-Primitive datatypes

Non-Primitive data types refer to objects and hence they are called **reference types.**Examples of non-primitive types include Strings, Arrays, Classes, Interface, etc. Below image depicts various non-primitive data types.



Let’s now understand these non-primitive data types in short.

**Strings:** String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object. If you wish to know more about Java Strings, you can refer to this article on [Strings in Java](https://www.edureka.co/blog/java-string/).

**Arrays:** Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index. If you wish to learn Arrays in detail, then kindly check out this article on [Java Arrays](https://www.edureka.co/blog/java-array/).

**Classes:**A [class in Java](https://www.edureka.co/blog/java-tutorial/#obj) is a blueprint which includes all your data.  A class contains fields(variables) and methods to describe the behavior of an object.

**Interface:** Like a class, an interface can have methods and variables, but the methods declared in [interface](https://www.edureka.co/blog/java-collections/#interface) are by default abstract (only method signature, no body).

So that was all about the non-primitive data types. Now let’s understand the difference between primitive and non-primitive data types.

## **Difference between primitive and non-primitive data types**

The  difference between **primitive** and **non-primitive** data types are as follows:

* Primitive types are predefined in [Java](https://www.edureka.co/blog/what-is-java/). Non-primitive types are created by the programmer and is not defined by Java.
* Non Primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type always has a value, whereas non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types start with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

# Java Tokens

In Java, the program contains classes and methods. Further, the methods contain the expressions and statements required to perform a specific operation. These statements and expressions are made up of tokens. In other words, we can say that the expression and statement is a set of tokens. The tokens are the small building blocks of a Java program that are meaningful to the [Java](https://www.javatpoint.com/java-tutorial) compiler. Further, these two components contain variables, constants, and operators. In this section, we will discuss what is tokens in Java.

**What is token in Java?**

The Java compiler breaks the line of code into text (words) is called Java tokens. These are the smallest element of the [Java program](https://www.javatpoint.com/java-programs). The Java compiler identified these words as tokens. These tokens are separated by the delimiters. It is useful for compilers to detect errors. Remember that the delimiters are not part of the Java tokens.

***token <= identifier | keyword | separator | operator | literal | comment***

## **Types of Tokens - Java token includes the following:**

* Keywords
* Identifiers
* Literals
* Operators
* Separators
* Comments

Keywords: These are the pre-defined reserved words of any programming language. Each [keyword](https://www.javatpoint.com/java-keywords) has a special meaning. It is always written in lower case. Java provides the following keywords:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| abstract | boolean | byte | break | class | native | new | package | private | protected |
| case | catch | char | continue | default | public | return | short | static | super |
| do | double | else | extends | final | switch | synchronized | this | thro | throws |
| finally | float | for | if | implements | transient | try | void | volatile | while |
| import | instanceof | int | interface | long | assert | const | enum | goto | strictfp |

Identifier: Identifiers are used to name a variable, constant, function, class, and array. It usually defined by the user. It uses letters, underscores, or a dollar sign as the first character. The label is also known as a special kind of identifier that is used in the goto statement. Remember that the identifier name must be different from the reserved keywords. There are some rules to declare identifiers are:

* The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
* The whitespace cannot be included in the identifier.
* Identifiers are case sensitive.

**Literals:** In programming literal is a notation that represents a fixed value (constant) in the source code. It can be categorized as an integer literal, string literal, Boolean literal, etc. It is defined by the programmer. Once it has been defined cannot be changed. Java provides five types of literals are as follows:

* Integer
* Floating Point
* Character
* String
* Boolean

|  |  |
| --- | --- |
| **Literal** | **Type** |
| 23 | int |
| 9.86 | double |
| false, true | boolean |
| 'K', '7', '-' | char |
| "Chaitanya" | String |
| null | any reference type |

**Operators:** In programming, operators are the special symbol that tells the compiler to perform a special operation. Java provides different types of operators that can be classified according to the functionality they provide. There are eight types of [operators in Java](https://www.javatpoint.com/operators-in-java), are as follows:

* Arithmetic Operators
* Assignment Operators
* Relational Operators
* Unary Operators
* Logical Operators
* Ternary Operators
* Bitwise Operators
* Shift Operators

|  |  |
| --- | --- |
| **Operator** | **Symbols** |
| Arithmetic | + , - , / , \* , % |
| Unary | ++ , - - , ! |
| Assignment | = , += , -= , \*= , /= , %= , ^= |
| Relational | ==, != , < , >, <= , >= |
| Logical | && , || |
| Ternary | (Condition) ? (Statement1) : (Statement2); |
| Bitwise | & , | , ^ , ~ |
| Shift | << , >> , >>> |

**Separators:** The separators in Java is also known as **punctuators**. There are nine separators in Java, are as follows:

separator <= ; | , | . | ( | ) | { | } | [ | ]

### Note that the first three separators (; , and .) are tokens that separate other tokens, and the last six (3 pairs of braces) separators are also known as delimiters. For example, Math.pow(9, 3); contains nine tokens.

* **Square Brackets []:** It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.
* **Parentheses ():** It is used to call the functions and parsing the parameters.
* **Curly Braces {}:** The curly braces denote the starting and ending of a code block.
* **Comma (,):** It is used to separate two values, statements, and parameters.
* **Assignment Operator (=):** It is used to assign a variable and constant.
* **Semicolon (;):** It is the symbol that can be found at end of the statements. It separates the two statements.
* **Period (.):** It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

**Comments:** [Comments](https://www.javatpoint.com/java-comments) allow us to specify information about the program inside our Java code. Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:

* **Line Oriented:** It begins with a pair of forwarding slashes (**//**).
* **Block-Oriented:** It begins with /\* and continues until it founds **\*/**.

# Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

## **Advantages**

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

## **Disadvantages**

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

# Static in java

 Static is used for memory management in java, when something is declared with static keyword it means and memory allocation is done only once for that element. Static keyword can be used for variables, method, blocks and class.

## **Static as Variable**

“static” can be used with variable, it means that variable belongs to class, Java static property is shared to all objects.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.
* It makes your program **memory efficient** (i.e., it saves memory).

## **Static method**

If you apply static keyword with any method, it is known as static method.

* 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.
* Static methods cannot be overridden, since they are resolved using static binding by the compiler at compile time. However, we can have the same name methods declared static in both superclass and subclass, but it will be called Method Hiding as the derived class method will hide the base class method.
* The static method cannot use non static data member or call non-static method directly.
* this and super cannot be used in static context.

**When to use static methods?**

* When you have code that can be shared across all instances of the same class, put that portion of code into static method.
* They are basically used to access static field(s) of the class.

**Instance Method-** Instance method are methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in which the method is defined.

**Memory allocation:** These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack. They can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depend on the access type provided to the desired instance method.

**Important Points:** Instance method(s) belong to the Object of the class, not to the class i.e. they can be called after creating the Object of the class.

Instance methods are not stored on a per-instance basis, even with virtual methods. They’re stored in a single memory location, and they only “know” which object they belong to because this pointer is passed when you call them.

They can be overridden since they are resolved using dynamic binding at run time.

**Instance method vs Static method**

* Instance method can access the instance methods and instance variables directly.
* Instance method can access static variables and static methods directly.
* Static methods can access the static variables and static methods directly.
* Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use this keyword as there is no instance for ‘this’ to refer to.

### **Why is the Java main method static?**

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

## **static block**

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### **Q) Can we execute a program without main() method?**

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

# this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

**Usage of Java this keyword**

Here is given the 6 usage of java this keyword.

1. **this can be used to refer current class instance variable.-** This keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
2. **this can be used to invoke current class method (implicitly)-** You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.
3. **this() can be used to invoke current class constructor.-** The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.
4. **this can be passed as an argument in the method call.-** This keyword can also be passed as an argument in the method. It is mainly used in the event handling. Application of this that can be passed as an argument:- In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.
5. **this can be passed as argument in the constructor call.-** We can pass this keyword in the constructor also. It is useful if we have to use one object in multiple classes.
6. **this can be used to return the current class instance from the method.-** We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive

# Object oriented programming concepts:

## **Inheritance in Java**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviour of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### **Types of inheritance in java**

**1. Single inheritance-** When a class inherits another class, it is known as a single inheritance.

**2. Multilevel inheritance-** When there is a chain of inheritance, it is known as multilevel inheritance.

**3. Hierarchical Inheritance -** When two or more classes inherits a single class, it is known as hierarchical inheritance

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

## **Aggregation in Java**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

### **Why use Aggregation?**

* For Code Reusability.

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

## **Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

**Real-life Illustration: Polymorphism -** A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Types of polymorphism**

* 1. Compile-time Polymorphism
  2. Runtime Polymorphism

**Type 1: Compile-time polymorphism -**It is also known as static polymorphism. This type of polymorphism is achieved by **function overloading or operator overloading.**

Note: But Java doesn’t support the Operator Overloading.

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

**Type 2: Runtime polymorphism**- It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by **Method Overriding**. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

### **Method Overloading in Java**

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**. If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs). Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

#### **Advantage of method overloading**

Method overloading increases the readability of the program.

#### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

##### **In Java, Method Overloading is not possible by changing the return type of the method only.**

##### 1) Method Overloading: changing no. of arguments

##### 2) Method Overloading: changing data type of arguments

##### **Q) Why Method Overloading is not possible by changing the return type of method only?**

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

###### **Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.**

##### **Can we overload java main() method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

#### **Method Overloading and Type Promotion**

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

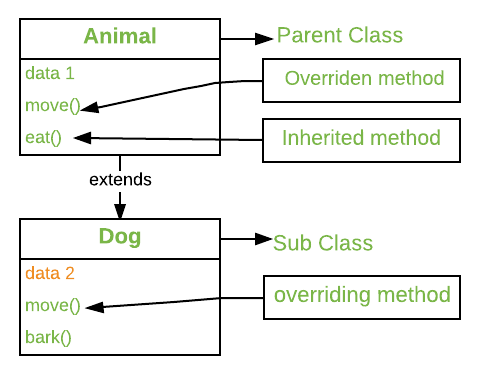


As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

##### **One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.**

### **Method Overriding in Java**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Method overriding is one of the way by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

#### **Usage of Java Method Overriding**

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**

1. Overriding and Access-Modifiers : The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.
2. Final methods cannot be overridden : If we don’t want a method to be overridden, we declare it as final. Please see Using final with Inheritance
3. Static methods can not be overridden(Method Overriding vs Method Hiding) : When you define a static method with same signature as a static method in base class, it is known as method hiding.
4. Private methods can not be overridden : Private methods cannot be overridden as they are bonded during compile time. Therefore we can’t even override private methods in a subclass.(See this for details).
5. The overriding method must have same return type (or subtype) : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomena is known as covariant return type.
6. Invoking overridden method from sub-class : We can call parent class method in overriding method using super keyword.
7. Overriding and constructor : We cannot override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
8. Overriding and Exception-Handling : Below are two rules to note when overriding methods related to exception-handling.

* Rule#1 : If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.
* Rule#2 : If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

1. Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.
2. Overriding and synchronized/strictfp method : The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it’s possible that a synchronized/strictfp method can override a non-synchronized/strictfp one and vice-versa.

#### **Why Method Overriding ?**

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Dynamic Method Dispatch is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

#### **When to apply Method Overriding ?(with example)**

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

#### **Difference between method overloading and method overriding in java**

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | Return type must be same or covariant in method overriding. |

#### **Dynamic Method Dispatch or Runtime Polymorphism in Java**

***Prerequisite: Overriding in java, Inheritance***

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden me

**In Java, we can override methods only, not the variables(data members), so runtime polymorphism cannot be achieved by data members.**

**Advantages of Dynamic Method Dispatch**

Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.

It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

### **Covariant Return Type**

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. *Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.*

#### **Advantages of Covariant Return Type**

Following are the advantages of the covariant return type.

1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.

2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.

3) Covariant return type helps in preventing the run-time *ClassCastExceptions* on returns.

Let's take an example to understand the advantages of the covariant return type.

### **Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

**Usage of Java super Keyword-**

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

### **Instance initializer block**

|  |
| --- |
| Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

Why use instance initializer block?

|  |  |
| --- | --- |
| Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. | |
| There are three places in java where you can perform operations: Method,constructor,block |
| it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below: | |

Note: The java compiler copies the code of instance initializer block in every constructor.

**Rules for instance initializer block** :

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

The instance initializer block is created when instance of the class is created.

The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).

The instance initializer block comes in the order in which they appear.

### final keyword in java**Final Keyword In Java**

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable-If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) Java final method-If you make any method as final, you cannot override it.

3) Java final class-If you make any class as final, you cannot extend it.

*Q) Is final method inherited?*

Ans) Yes, final method is inherited but you cannot override it. For Example:

*Q) What is blank or uninitialized final variable?*

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

*Que) Can we initialize blank final variable?*

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

Q) Can we declare a constructor final?

No, because constructor is never inherited.

### **Static Binding and Dynamic Binding**

Connecting a method call to the method body is known as binding. There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

### **Static binding**

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

### **Dynamic binding**

When type of the object is determined at run-time, it is known as dynamic binding.

class Animal {  
 void eat() {  
 System.*out*.println("animal is eating...");  
 }  
}  
class Dog extends Animal {  
 void eat() {  
 System.*out*.println("dog is eating...");  
 }  
 public static void main(String args[]) {  
 Animal a = new Dog();  
 a.eat();  
 }  
}

## **Abstraction in Java**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

*Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.*

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

### **Abstract classes and Abstract methods :**

* An abstract class is a class that is declared with an abstract keyword.
* An abstract method is a method that is declared without implementation.
* An abstract class may or may not have all abstract methods. Some of them can be concrete methods
* A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make the subclass itself abstract.
* Any class that contains one or more abstract methods must also be declared with an abstract keyword.
* There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
* An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

### **Abstract Class in java-** Following are some important observations about abstract classes in Java.

* An instance of an abstract class cannot be created.
* Constructors are allowed.
* We can have an abstract class without any abstract method.
* There can be final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simper terms final method cannot be abstract itself as it will yield error: “Illegal combination of modifiers: abstract and final”
* We are not allowed to create object for any abstract class.
* We can define static methods in an abstract class
* We can use abstract keyword for declaring top level classes (Outer class) as well as inner classes as abstract
* If a class contain at least one abstract method then compulsory we should declare class as abstract
* If Child class is unable to provide implementation to all abstract methods of Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to remaining abstract method

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember-

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

### **Abstract class having constructor, data member and methods**

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

* **Rule: If there is an abstract method in a class, that class must be abstract.**
* **Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.**

### **Few Observations about abstract class-**

1. Observation 1: instance of an abstract class cannot be created; we can have references to abstract class type though.
2. Observation 2: abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created.
3. Observation 3: In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited.
4. Observation 4: Abstract classes can also have final methods (methods that cannot be overridden)
5. Observation 5: For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.
6. Observation 6: Similar to the interface we can define static methods in an abstract class that can be called independently without an object.
7. Observation 7: We can use abstract keyword for declaring top level classes (Outer class) as well as inner classes as abstract
8. Observation 8: If a class contain at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error because If a class contains at least one abstract method then implementation is not complete for that class and hence it is not recommended to create a object so in order to restrict object creation for such partial classes we use abstract keyword
9. Observation 9: If Child class is unable to provide implementation to all abstract methods of Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to remaining abstract method