**Core Java Differences**

**Variables & types**

|  |  |  |
| --- | --- | --- |
| **Local** | **Instance** | **Static** |
| declared inside method, constructor, or block | declared in a class, but outside a method | declared in a class, but outside a method. Must be declared static keyword |
|  | Each object will have the copy of instance variable, it won’t reflect on other objects | If any object changes the value of static variable it will retain the changed value |
| Created when method or constructor is entered. | Created when instance of class is created with new. | Created when the program starts. |
| Local variables (including formal parameters) are visible only in the method, constructor, or block in which they are declared.  Access modifiers (private, public, ...) *cannot* be used with local variables. | Instance (field) variables can been seen by all methods in the class. | Same as instance variable, but are often declared public to make constants available to users of the class. |
| Initialisation of local variable is mandatory before the first use. | Zero for numbers, false for boolean, or null for object references. May be assigned value at declaration or in constructor. | Same as instance variable, and can be assigned value in special *static initializer block*. |
|  | Instance Variable can be accessed only by creating objects.  **ObjectName.variablename;** | To access static variables, we need not create an object of that class, we can simply access the variable as,  **ClassName.variablename;**  **Note:-** Compiler will automatically append the class name. |

**Instance variable Vs Static variable**

* Each object will have its **own copy** of instance variable whereas we can only have **one copy** of a static variable per class irrespective of how many objects we create.
* Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of instance variable. In case of static, changes**will be reflected** in other objects as static variables are common to all object of a class.
* We can access instance variables **through object references** and Static Variables can be accessed **directly using class name.**

**Datatypes & types**

|  |  |
| --- | --- |
| **Primitive** | **Non – Primitive(Wrapper classes)** |
| A primitive data type specifies the size and type of variable values, and it has no additional methods. | Non-primitive data types are called **reference types** because they refer to objects. |
| There are eight primitive data types in Java. byte, short, int, long, float, double, boolean and char | [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), Integer, Float, Double etc |
| Primitive type has always a value | Non-primitive types can be null |
| Primitive type starts with a lowercase letter | Non-primitive types starts with an uppercase letter |

**Use float or double?**

The precision of a floating-point value indicates how many digits the value can have after the decimal point. The precision of **float is only six or seven decimal digits**, while **double variables have a precision of about 15 digits**. Therefore, it is safer to use double for most calculations.

**Static methods & Non static Methods**

|  |  |
| --- | --- |
| **Static methods** | **Non static Methods** |
| A **static method** is a method that belongs to a class, but its not belongs to an instance of that class and this method can be called without the instance or object of that class. | Every methods in java are non-static method, but the methods **must not have** **static** keyword before method name. **non-static** methods can access any **static** method and **static** variable also, without using the object of the class. |
| Method can **only access only static data members and static methods** of another class or same class but cannot access non-static methods and non-static variables. | In **non-static** method, the method can **access static data members and static methods** as well as **non-static members and method of another class** or same class. |
| In **static** method, less memory is use for execution because memory allocation happens only once, because the static keyword fixed a particular memory for that method in ram. | In **non-static** method, much memory is used for execution because here memory allocation happens when the method is invoked and the memory is allocated every time when the method is called. So much memory is used here. |
| Invoked by,  **ClassName.methodName;** | Invoked by,  **objectName.methodName();** |

**Constructor & Method**

|  |  |
| --- | --- |
| **Constructor** | **Method** |
| Constructor is used to initialize an object | method is used to define the functionality of an object |
| Constructor should be of the same name as that of class | Method name should not be of the same name as that of class |
| Constructor does not return any value | Method may/may not return a value. |
| In case constructor is not present, a default constructor is provided by java compiler | In the case of a method, no default method is provided. |
| Constructors are invoked implicitly | Methods are invoked explicitly |
| Syntax,  accessmodifier className() {  //constructor body  } | Syntax,  accessmodifier(optional) nonaccessmodifier(optional) returnType methodName(parameterList) {  //method body  } |

**this & super keywords**

|  |  |
| --- | --- |
| **This** | **Super** |
| **this** is used to refer **current-class’s instance as well as static members** | **super** is used to refer **super-class’s instance as well as static members** |
| We can access the fields and methods of **current** class | We can access the fields and methods of **parent** class in any child class. |
| this() should be **first statement** inside constructor | super() should be **first statement** inside constructor |
| **private, default, public** and **protected** **methods**can be called by the **this**keyword | Only **public** and **protected** **methods**can be called by the **super**keyword |

Note :-

We can use **either super() or this() as first statement inside constructor and not both**.  
Both **this and super are non static** and **cannot be used in static context**, which means you cannot use this and super keyword inside main method in Java.

**Method overloading and Method overriding**

|  |  |
| --- | --- |
| **Method overloading** | **Method overriding** |
| Happens at [compile-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) | Happens at [run-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) |
| It is used to increase readability of the program | It is used for the specific implementation of a method already provided by the super class |
| Static methods can be overloaded | Static methods can’t be overrided |
| Done in the same class | Base and child classes are required |
| private and final methods can be overloaded | private and final methods can’t be overrided |
| Return type of method does not matter | Return type should be same |
| Argument list should be different | Argument list should be same |

**Abstract class and interface**

|  |  |
| --- | --- |
| **abstract class** | **Interface** |
| Abstract class achieves partial abstraction (**0 to 100%)** | Interface achieves fully abstraction (**100%**) |
| The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface |
| Abstract class can **have abstract and non-abstract** methods | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| Doesn’t support multiple inheritance. | Support multiple inheritance. |
| A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| Abstract class can have **final, non-final, static and non-static variables** | Interface has **only static and final variables** |
| Abstract class can provide the implementation of interface | Interface can’t provide the implementation of abstract class |
| Abstract class can be extended using keyword “extends” | A Java interface can be implemented using keyword “implements” |
| An abstract class can extend another Java class and implement multiple Java interfaces | An interface can extend another Java interface only |

**throw & throws**

|  |  |
| --- | --- |
| **Throw** | **Throws** |
| keyword is used to throw an exception explicitly | Keyword used to declare an exception, |
| **throw** is followed by an instance of Exception class | **throws** is followed by exception class names |
| **Used within the method** | **Used with the method signature** |
| can throw one exception at a time | can handle multiple exceptions by declaring them using throws keyword. |
| Example:-  void checkAge(int age){  if(age<18)  throw new ArithmeticException("Not Eligible for voting");  else  System.out.println("Eligible for voting");  } | Example:-  void division(int a, int b) throws ArithmeticException{  int t = a/b;  System.out.println("Division result:”+t);  } |

**final, finally & finalize**

|  |  |  |
| --- | --- | --- |
| **final** | **Finally** | **finalize** |
| final is a keyword | finally is a block | finalize is a method |
| **final** keyword can be used with class method and variable | Finalize is used to place important code, it will executed whether exception handled or not | Finalize is a protected non-static method that is defined in the **Object** class and thus is available for any and all the objects in Java and this method is called by the garbage collector before an object is completely destroyed |
| A final class cannot be instantiated, a final method cannot be overridden and a final variable cannot be reassigned. |  | This method is declared as protected to restrict its use from outside the class. But you can override it from within the class to define its properties at the time of garbage collection. |

**Checked Exception & Unchecked Exception**

|  |  |
| --- | --- |
| **Checked Exception** | **Unchecked Exception** |
| Checked exceptions are checked at compile-time. | Unchecked exceptions are not checked at compile time |
| If a method is throwing a checked exception then it should handle the exception using [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](https://beginnersbook.com/2013/04/java-throws/) | All Unchecked exceptions are direct sub classes of **RuntimeException** class |
| Checked exceptions are forced by compiler and used to indicate exceptional conditions that are out of the control of the program | Unchecked exceptions are occurred during runtime and used to indicate programming errors |
| Example,  FileNotFoundException | Examples, NullPointerException, IllegalArgumentExceptions, ArrayIndexOutOfBoundsException |

**----\*----\*----**