1. **Method Signature**

* a method signature is part of the method declaration. It's the combination of the method name and the parameter list.

More generally, method declarations have six components, in order:

* Modifiers—such as public, private, and others you will learn about later.
* The return type—the data type of the value returned by the method, or void if the method does not return a value.
* The method name—the rules for field names apply to method names as well, but the convention is a little different.
* The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
* The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here

1. **Over Load Main Method**

* We can overload the main method in java
* but JVM only calls the original main method, it will never call our overloaded main method**.**
* Overloading takes place at compile time
* overriding takes place at runtime
* [JVM](https://javarevisited.blogspot.com/2019/04/top-5-courses-to-learn-jvm-internals.html)will always call this main method, no matter how many overloaded main methods you will put on this class

1. **Super keyword**

* the super keyword refers to superclass (parent) objects.
* It is used to call superclass methods, and to access the superclass constructor
* The most common use of the super keyword is to eliminate the confusion between super classes and subclasses that have methods with the same name

**Use of super with variables:**

* This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM

**Use of super with methods:**

* This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword

1. **Object Super Class**

* Every class in Java is directly or indirectly derived from the Object class.
* If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived.
* Object class methods are available to all Java classes
* Object class acts as a root of inheritance hierarchy in any Java Program.
* There are methods in Object class:

**toString()** : toString() provides String representation of an Object and used to convert an object to String.

* The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance,
* the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object.
* **hashCode()**: For every object, JVM generates a unique number which is hashcode.
* It returns distinct integers for distinct objects.
* A common misconception about this method is that hashCode() method returns the address of object, which is not correct.
* It convert the internal address of object to an integer by using an algorithm.
* The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.
* **equals(Object obj)**: Compares the given object to “this” object (the object on which the method is called).
* It gives a generic way to compare objects for equality.
* It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects.
* **getClass()** : Returns the class object of “this” object and used to get actual runtime class of the object.
* It can also be used to get metadata of this class.
* The returned Class object is the object that is locked by static synchronized methods of the represented class.
* As it is final so we don’t override it.
* **finalize()** method : This method is called just before an object is garbage collected.
* It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object.
* We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks.
* **clone()** : It returns a new object that is exactly the same as this object
* **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
* **notifyAll()-**It wakes up all the threads that called wait() on the same object.

1. **Difference between method overloading,Method overriding**

|  |  |
| --- | --- |
| **Method overloading** | **Method overriding** |
| The argument type needs to be different in Method Overloading (at least the order).  You can use any access modifier, or it can be different.  A user can generally perform method overloading within the same class.  It is also known as the early binding, static polymorphism, or compile-time polymorphism.  The method overloading assists in raising the program’s readability  It may or may not be requiring inheritance.  The parameter needs to be different in the case of method overloading | The argument type needs to be the same in Method Overriding (including the order).  The access modifier for a subclass method must be the very same or higher than the access modifier of the superclass method  A user can usually perform the method overriding in two of the classes through the Inheritance (considered an Is-A relationship).  It is also known as late binding, dynamic polymorphism, or runtime polymorphism.  It assists in granting the specific implementation of any method (that the superclass or parent class provides)  It is always in need of inheritance  The parameter needs to be the same in the case of method overriding |

**Protected modifier**

* The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected
* The protected access modifier is accessible within package and outside the package but through inheritance only**.**
* It provides more accessibility than the default modifier
* Protected data members are accessible within the class and outside the class Within same package or package of subclasses.

**Var arguments**

* Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments.
* Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

■ **Var-arg type** When you declare a var-arg parameter, you must specify the type of the argument(s) this parameter of your method can receive. (This can be a primitive type or an object type.)

■ **Basic syntax** To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space, and then the name of the array that will hold the parameters received

■ Other parameters It's legal to have other parameters in a method that uses a var-arg.

■ **Var-args limits The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method**.

Let's look at some legal and illegal var-arg declarations:

**Legal:**

void doStuff(int... x) { } // expects from 0 to many ints // as parameters

void doStuff2(char c, int... x) { } // expects first a char, // then 0 to many ints void doStuff3(Animal... animal) { } // 0 to many Animals

**Illegal:**

void doStuff4(int x...) { } // bad syntax

void doStuff5(int... x, char... y) { } // too many var-args

void doStuff6(String... s, byte b) { } // var-arg must be last

**Covariant return Type**

* When a subclass wants to change the method implementation of an inherited method (an override),
* the subclass must define a method that matches the inherited

version exactly. Or,

* as of Java 5, you're allowed to change the return type in the overriding method as long as the new return type is a subtype of the declared return type of the overridden (superclass) method.

Let's look at a covariant return in action:

class Alpha

{

Alpha doStuff(char c)

{

return new Alpha();

}

}

class Beta extends Alpha

{

Beta doStuff(char c)

{ // legal override in Java 1.5

return new Beta();

}

}

**Rules for Covariant Return Type in Java**

1. The return type of overriding method in the subclass should be either the same as the return type of superclass or subclass.

2. The covariant return type is applicable only for object types not for primitive types.