Differences-

throw	**throws**
Java throw keyword is used to throw an exception explicitly in the code, inside the function or the block of code. The Exception may or may not be user-defined.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
The throw keyword is followed by an instance of Exception to be thrown. Eg-void m1(){ throw new ArithmeticException();}	The throws keyword is followed by class names of Exceptions to be thrown. Eg- void m1()throws ArithmeticException(){}
throw is used within the method .	throws is used with the method signature.
We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.
Statement written after throw is unreachable.	There is no such rule while using throws.

Exception	**Error**
We can recover from exceptions by handling them using try-catch block.	Recovering from Error is not possible.
Exceptions include both checked as well as unchecked type.	All errors in java are unchecked type.
Program(or Programmer) itself is responsible for causing exceptions.	Errors are mostly caused by the environment in which program is running.
They are defined in java.lang.Exception package.	They are defined in java.lang.Error package.
Examples : Checked Exceptions : IOException, Unchecked Exceptions : ArrayIndexOutOfBoundException,	Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError
All exceptions occurs at runtime but checked exceptions are known to the compiler while unchecked are not.	It occurs at run time,(Not considering compile error/syntax error),if we consider Exception hierarchy.

Checked Exception	**Unchecked Exception**
They occur at compile time.	These exceptions occur at runtime.
The compiler checks for a checked exception.	The compiler doesn't check for these kinds of exceptions.
These exceptions can be handled at the compilation time.	These kinds of exceptions can't be caught or handled during compilation time.
Everything in Throwable class except Error class and RuntimeException class are Checked Exception.	In java Exceptions ,under *Error* and *RuntimeException* classes are unchecked exceptions.

Fully Checked Exception	**Partially Checked Exception**
A Checked Exception is said to be fully checked exception if and only if all its child classes also checked.	A Checked Exception is said to be partially Checked exception if and only if some of its child classes are Unchecked.
Example- IOException, InterruptedException	Example- Throwable, Exception

---

Method Overloading	**Method Overriding**
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *method signature must be different*.	In case of method overriding, *method signature must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
In method overloading, the return type can or can not be the same.	In method overriding, the return type must be the same or co-variant.
Method resolution is taken care by compiler based on object reference.	Method resolution is taken care by JVM based on object type.

Method Hiding	Method Overriding
Both methods must be static.	Both methods must be non-static.
Method resolution takes care by the compiler based on the reference type.	Method resolution takes care by JVM based on runtime object.
It is considered as compile-time polymorphism or static polymorphism or early binding.	It is considered as runtime polymorphism or dynamic polymorphism or late binding.
Method hiding can be defined as, "if a subclass defines a static method with the same signature as a static method in the super class, in such a case, the method in the subclass hides the one in the superclass."	**Method overriding** means subclass had defined an instance method with the same signature and co variant return type as the instance method in the superclass.

String	StringBuffer
The String class is immutable.	The StringBuffer class is mutable.
String class uses String constant pool and heap memory.	StringBuffer uses Heap memory.
String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
Consumes more memory when we concatenate too many strings because every time it creates new instance.	Consumes less memory when we concatenate strings.
It overrides both equal() and hashcode() techniques of object class.	It cannot override equal() and hashcode() methods.
Methods are not synchronized	All methods are synchronized in this class.
During Threading, it is Fast.	During Threading, it is Slow.

StringBuffer	StringBuilder
Thread-Safe	Not Thread-Safe
Synchronised	Not Synchronised
Slower	Faster
Since Java 1.0	Since Java 1.5

Abstraction	Data Hiding
Abstraction is the process of hiding certain details and showing only essential information to the user.	Variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.
This is usually achieved using ‘abstract’ class concept, and by implementing interfaces.	This can be achieved using access specifiers, such as ‘private’, and ‘protected’.
The abstraction's purpose is to hide the complex implementation details of the program or software	On the other hand, data hiding is implemented to achieve encapsulation.
Abstraction helps to reduce the complexity of the system.	**Data hiding secures the data members.**

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Wrapping of data and functions of class together is Encapsulation. Encapsulation is Data Hiding and Abstraction.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
It focuses on the **external** lookout.	It focuses on **internal** working.

Final	**Finalize**	**Finally**
final is the keyword ,non access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependent on the exception.	finalize method is executed just before the object is destroyed by garbage collector.

Different ways **toString()** works in java-

1)Printing user-defined class reference

```
public class Test {
    public static void main(String[] args) {
        Test t=new Test();
        System.out.println(t);
    }
}
```

Output:-
Test@76ed5528

Internal toString() Implementation in Object.class -

```
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
```

2) Printing Exception Reference

```
public class Test {
    public static void main(String[] args) {
        Exception e=new Exception();
        System.out.println(e.toString());
        Exception e1=new Exception("hey");
        System.out.println(e.toString());}}
```

Output-
java.lang.Exception
java.lang.Exception

Internal Implementation of toString() in Throwable.class-

```
public String toString() {
        String s = getClass().getName();
        String message = getLocalizedMessage();
        return (message != null) ? (s + ": " + message) : s;
    }
```

3)Printing Thread Reference

```
public class Test {
    public static void main(String[] args) {
       Thread t=new Thread();
       System.out.println(t);
    }
}
```

Output-
Thread[Thread-0,5,main]

Internal Implementation of toString() in Thread.class-

```
   public String toString() {
        ThreadGroup group = getThreadGroup();
        if (group != null) {
            return "Thread[" + getName() + "," + getPriority() + "," +
                           group.getName() + "]";
        } else {
            return "Thread[" + getName() + "," + getPriority() + "," +
                            "" + "]";
        }
    }
```

4) Printing String Reference

```
public class Test {
    public static void main(String[] args) {
      String s="hi";
      System.out.println(s.toString());
      String s1="hi1";
      System.out.println(s1.toString());
    }
}
```

Output-
hi
hi1

Internal Implementation of toString() in String.class –

```
 public String toString() {
        return this;
    }
```

5) Printing Array Reference

```
public class Test {
    public static void main(String[] args) {
    int arr[]={1,2,3};
    System.out.println(arr.toString());
    System.out.println(arr.getClass().getName());
    int arr1[]={1,2,3};
    System.out.println(arr1.toString());
    System.out.println(arr1.getClass().getName());
    }
}
```

Output-

```
[I@76ed5528
[I
[I@2c7b84de
[I
```

Internal Implementation of toString() in Object.class –

```
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
```

Sr. No.	Key	==	equals() method
1	**Type**	== is an operator.	equals() is a method of Object class.
2	**Comparision**	== should be used during reference comparison. == checks if both references points to same location or not.	equals() method should be used for content comparison. equals() method evaluates the content to check the equality.

3	**Object**	== operator can not be overriden.	equals() method if not present and Object.equals() method is utilized, otherwise it can be overridden.

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
Garbage Collection	C++ does not support garbage collection.	Java supports garbage collection.
Compilation and Interpretation	C ++ is only compiled and cannot be interpreted.	Java is both compiled and interpreted.

Aggregation vs Composition

Composition and **Aggregation** are the two forms of association.

1. Dependency: Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don't exist separate to a Human

2. Type of Relationship: Aggregation relation is **"has-a"** and composition is **"part-of"** relation.

3. Type of association: Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Aggregation	Composition
Aggregation can be described as a "Has-a" relationship, which denotes the association between objects.	Composition means one object is contained in another object. It is a special type of aggregation (i.e. Has-a relationship), which implies one object is the owner of another object, which can be called an ownership association.
There is mutual dependency among objects.	There is a unidirectional relationship, this is also called "part of" relationship.
It is a weak type of association, both objects have their own independent lifecycle.	It is a strong type of association (aggregation), the child object does not have its own life cycle.
The associated object can exist independently and have its own lifecycle.	The child's life depends upon the parent's life. Only the parent object has an independent lifecycle.
UML representation of White Diamond denotes aggregation.	UML representation of Black Diamond denotes composition.
For example, the relationship between a student and a department. The student may exist without a department.	For example, a file containing in a folder, if the folder deletes all files inside will be deleted. The file can not exist without a folder.