Task 3:

**Object class & methods in object class with descriptions and syntax**

# Object class in Java

Object is a super class of all classes in java.

Any class that you write by default extends object.

The reason begin, Java plays over objects which internally means the object.

This object is used to perform multiple operations

1. To create a object,

How it helps by using hashCode method, which represents the address of the object that has been created in heap memory.

1. Comparing Data,

When we want to check 2 objects if they are equal or not we use equals method. Internal implementation is just comparing the hashCode, that is the reason why it is recommended to override equals and hashCode in your implemented class to have specific implementation.

1. Wait

If we want a object to stop executing for a period of time ir till someone calls it, this method will be used.

1. Notify

To notify a thread.

1. toString to display the content of the class.

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object.

For **Example**:

Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



### Methods of Object class:

### The Object class provides many methods. They are as follows:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashCode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

**What is the syntax of class and object?**

Syntax: **ClassName ObjectName;** Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object.

# Object Cloning in Java:

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

**protected** Object clone() **throws** CloneNotSupportedException

### Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

## **The clone() Method**

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:

***aCloneableObject*.clone();**

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as

protected Object clone() throws CloneNotSupportedException

**or:**

public Object clone() throws CloneNotSupportedException

if you are going to write a clone() method to override the one in Object.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

## **The equals() Method**

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object *references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

public class Book {

String ISBN;

public String getISBN() {

return ISBN;

}

public boolean equals(Object obj) {

if (obj instanceof Book)

return ISBN.equals((Book)obj.getISBN());

else

return false;

}

}

Consider this code that tests two instances of the Book class for equality:

// Swing Tutorial, 2nd edition

Book firstBook = new Book("0201914670");

Book secondBook = new Book("0201914670");

if (firstBook.equals(secondBook)) {

System.out.println("objects are equal");

} else {

System.out.println("objects are not equal");

}

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the equals() method if the identity operator is not appropriate for your class.

**Note:** If you override equals(), you must override hashCode() as well.

## **The finalize() Method**

The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, don't rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors. Instead, use a try-with resources statement to automatically close your application's resources. See [The try-with-resources Statement](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html" \t "https://docs.oracle.com/javase/tutorial/java/IandI/_top) and [Finalization and Weak, Soft, and Phantom References](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/considerations.html" \l "sthref63) in Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide.

## **The getClass() Method**

You cannot override getClass.

The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:

void printClassName(Object obj) {

System.out.println("The object's" + " class is " +

obj.getClass().getSimpleName());

}

The [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html" \t "https://docs.oracle.com/javase/tutorial/java/IandI/_blank) class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

## **The hashCode() Method**

The value returned by hashCode() is the object's hash code, which is an integer value generated by a hashing algorithm.

By definition, if two objects are equal, their hash code *must also* be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

## **The toString() Method**

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

System.out.println(firstBook.toString());

which would, for a properly overridden toString() method, print something useful, like this:

ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

# **Writing Final Classes and Methods**

You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

class ChessAlgorithm {

enum ChessPlayer { WHITE, BLACK }

...

**final** ChessPlayer getFirstPlayer() {

return ChessPlayer.WHITE;

}

...

}

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.