Chapter 15

Garbage Collection

Introduction

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.
- In java, garbage collection is performed automatically. So, java provides better memory management.
    - **finalize() method**
        - The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.
        - This method is defined in Object class as:
            - protected void finalize(){}
        - **Note:** The Garbage collector of JVM collects only those objects that are created by new keyword. So if any object is created without new, use the finalize() method to perform cleanup processing (destroying remaining objects).
    - **gc() method**
        - The gc() method is used to invoke the garbage collector to perform cleanup processing.
        - The gc() is found in System and Runtime classes.
            - public static void gc(){}
        - **Note:** Garbage collection is performed by daemon thread called Garbage Collector (GC). This thread calls the finalize() method before object is garbage collected.

How Java Garbage Collection Works

- Java garbage collection is an automatic process.
- The programmer does not need to explicitly mark objects to be deleted.
- Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification.

- The motivation behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.
- All its garbage collectors follow the same basic process.
    - **Step 1:** *unreferenced objects* are identified and marked as ready for garbage collection.
    - **Step 2:** marked objects are deleted.
    - Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap.
    - The compaction process makes it easier to allocate memory to new objects sequentially after the block of memory allocated to existing objects.

Java (JVM) Memory Model

- Java objects are created in Heap and timely switched from one part to another called generation.
- This switching of objects within heap memory depends on the object's age.
- The heap memory area in the JVM is divided into three sections:
    - *Young Generation*
    - *Old Generation*
    - *Permanent Generation*

- **Young Generation**:
    - The young generation is the place where all the new objects are created.
    - Young Generation is divided into three parts:
        - *Eden memory space*
        - *Survivor 0 (S0) memory space*
        - *Survivor 1 (S1) memory space*
    - Most of the newly created objects are located in the ***Eden*** memory space.
    - When ***Eden*** space is filled with objects, garbage collection is performed , which is called Minor GC all the survivor objects are moved to one of the survivor spaces, say **Survivor 0 (S0)**.
    - Minor GC also checks the survivor objects in **Survivor 0 (S0)** and move them to the other survivor space, say **Survivor 1 (S1)**. So that at a time, one of the survivor space is always empty.
    - Objects that are survived after many cycles of GC, are moved to the ***Old generation*** memory space.
        - Usually, it's done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

- **Old Generation:**
  - Objects that are long-lived are eventually moved from the Young Generation to the Old Generation.
  - When objects are garbage collected from the Old Generation, it is a major garbage collection event.

- **Permanent Generation**
  - Permanent Generation or "Perm Gen" contains the application metadata required by the JVM such as classes and methods.
  - Perm Gen also contains Java SE library classes and methods.
  - Classes that are no longer in use may be garbage collected from the Permanent Generation.
  - Note that Perm Gen is not part of Java Heap memory.
  - Perm Gen objects are garbage collected in a full garbage collection.

Oracle's HotSpot

- Although there are many JVMs, Oracle's HotSpot is by far the most common.
- All of HotSpot's garbage collectors implement a generational garbage collection strategy that categorizes objects by age.
- HotSpot has four garbage collectors that are optimized for various use cases.
  - *Serial*
  - *Parallel*
  - *CMS (Concurrent Mark Sweep)*
  - *G1 (Garbage First)*

- **Serial:**
  - All garbage collection events are conducted serially in one thread.
  - Compaction is executed after each garbage collection.

- **Parallel:**
  - Multiple threads are used for minor garbage collection.
  - A single thread is used for major garbage collection and Old Generation compaction.
  - Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.

- **CMS (Concurrent Mark Sweep):**
  - Multiple threads are used for minor garbage collection using the same algorithm as Parallel.
  - Major garbage collection is multi-threaded, like Parallel Old, but CMS runs concurrently alongside application processes to minimize "stop the world" events (i.e. when the garbage collector running stops the application).
  - No compaction is performed.

- **G1 (Garbage First):**
  - The newest garbage collector is intended as a replacement for CMS.
  - It is parallel and concurrent like CMS, but it works quite differently under the hood compared to the older garbage collectors.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

Making Objects Eligible for Garbage Collection

- If the reference variable of the object is no longer available; Objects that fall under this category are also termed as *unreachable objects*.
- An object that does not have a reference is essentially not present on the memory heap, which means that it is not available for use and cannot add any value. Figure below shows the unreachable and reachable objects.
- Only objects that no longer have a reference associated with them can be deemed or made eligible for garbage collection.

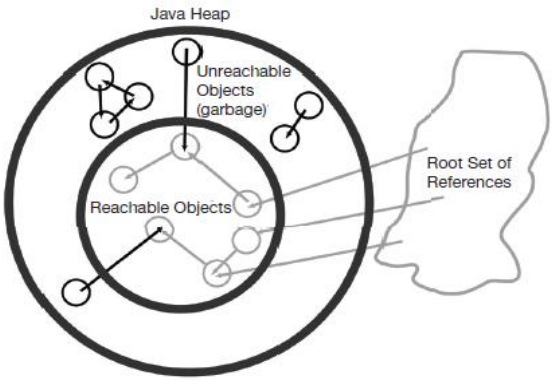

Figure: Reachable and unreachable objects in Java heap.

- **Example:** Program to demonstrate that objects that are created within the scope of a method will be deemed useless after execution of that method is complete.

```
public class UnreachableObjectsExample {

        private String myObject;

        public static void main(String args[]) {
```

```
        // Executing testMethod1 method
        testMethod1();

        // Requesting garbage collection
        System.gc();
    }

    public UnreachableObjectsExample(String myObject) {
        this.myObject = myObject;
    }

    private static void testMethod1() {
        // After existing testMethod1(), the object myObjectTest1 becomes
        //unreachable
        UnreachableObjectsExample myObjectTest1 = new
                                UnreachableObjectsExample("myObjectTest1");
        testMethod2();
    }

    private static void testMethod2() {
        // After existing testMethod2(), the object myObjectTest2 becomes
        //unreachable
        UnreachableObjectsExample myObjectTest2 = new
                                UnreachableObjectsExample("myObjectTest2");
    }
    @Override
    protected void finalize() throws Throwable {

        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is successful for " +
                                                        this.myObject);
    }
}
```

Output:

```
Garbage collection is successful for myObjectTest2
Garbage collection is successful for myObjectTest1
```

Note:

- Since both the objects within the method had become unreachable, hence the output as above.
- The output shows that any object within a method becomes useless after execution of the method; the object automatically becomes eligible for garbage collection.

Java Cleaners: The Modern Way to Manage External Resources

- Prior to Java 9 programmers could use a finalizer by overriding the Object's class finalize() method.
- Finalizers have many disadvantages, including being slow, unreliable and dangerous.
- Since Java 9, Finalizers have been deprecated and programmers have a better option to achieve this in Cleaners.
- Cleaners provide a better way to manage and handle cleaning/finalizing actions.
- Cleaners work in a pattern where they let resource holding objects register themselves and their corresponding cleaning actions.
- And then Cleaners will call the cleaning actions once these objects are not accessible by the application code.

Finalizers Vs Cleaners

Finalizers	Cleaners
Finalizers are invoked by one of Garbage Collector's threads, you as a programmer don't have control over what thread will invoke your finalizing logic	Unlike with finalizers, with Cleaners, programmers can opt to have control over the thread that invokes the cleaning logic.
Finalizing logic is invoked when the object is actually being collected by GC	Cleaning logic is invoked when the object becomes ***Phantom Reachable***, that is our application has no means to access it anymore
Finalizing logic is part of the object holding the resources	Cleaning logic and its state are encapsulated in a separate object.
No registration/deregistration mechanism	Provides means for registering cleaning actions and explicit invocation/deregistration

Example:

```
import java.lang.ref.Cleaner;

public class UnreachableObjectsExample {

    private String myObject;

    //To obtain a CLEANER instance, call the static create() method on the Cleaner class:
    private static final Cleaner CLEANER = Cleaner.create();
```

```
//Cleanable represents an object and a cleaning action registered in a Cleaner.
private static Cleaner.Cleanable cleanable;

//constructor
public UnreachableObjectsExample(String myObject) {
        this.myObject = myObject;
        System.out.println("The Object is " + this.myObject);

        // create a Cleanable instance, which will help us register the cleaning actions
        // against my object
        // The register() method of a cleaner takes two arguments, the object it is
        //supposed to monitor for cleaning, and the action to perform for cleaning.
       cleanable = CLEANER.register(this, () -> System.out.println("Cleaning action"));
}

//testMethod1() method
private static void testMethod1() {

        // The new object, such as "myObjectTest1" is created and display content
        // After existing testMethod1(), the object myObjectTest1 becomes unreachable
        UnreachableObjectsExample myObjectTest1 = new
                                                UnreachableObjectsExample("ABC");
        System.out.println("myObjectTest1: "+ myObjectTest1);
        testMethod2();
}

//testMethod2() method
private static void testMethod2() {

        // The new object, such as "myObjectTest2" is created and display content
        // After existing testMethod2(), the object myObjectTest2 becomes unreachable
        UnreachableObjectsExample myObjectTest2 = new
                                                UnreachableObjectsExample("XYZ");
        System.out.println("myObjectTest2: "+ myObjectTest2);
}

// close method
public static void close() {

        //Unregisters the cleanable and invokes the cleaning action.
        //The cleanable's cleaning action is invoked at most once
        //regardless of the number of calls to clean.
        cleanable.clean();
}
```

```
        // main method
        public static void main(String args[]) {

                // Executing testMethod1() method
                testMethod1();

                //Executing close() method
                close();
        }
}
```

Output:

```
The Object is ABC
myObjectTest1: UnreachableObjectsExample@4f023edb
The Object is XYZ
myObjectTest2: UnreachableObjectsExample@3a71f4dd
Doing cleaning
```

Note:

- Take the following line taken from the above program

```
cleanable = CLEANER.register(this, () -> System.out.println("Cleaning action"));
```

- The above line can be replaced with following code

```
cleanable = CLEANER.register(myObject, new State());
```

- Furthermore, add the following method

```
private static class State implements Runnable {
        public void run() {
        System.out.print("Cleaning action");
        }
}
```

How can an object be unreferenced?

- An object can be unreferenced basically by 3 ways:
  - By assigning a reference to another
  - By nullified the reference
  - By anonymous object

- **Assigning a reference to another**
  - Reference IDs are extremely important in Java and help in addressing each of the objects and variables that are being used in any code.

- In case, one object's reference ID is used to refer to other object's reference ID, then the first object that was initially being referenced becomes unreachable and cannot be used in the program or code in any way. Hence, the first object is deemed eligible for garbage collection.
- **Example:** The following program is an example of a situation where a reference ID is used to reference multiple objects.

```
public class ReassigningReferenceExample {

        private String myObject;

        public ReassigningReferenceExample(String myObject) {
                this.myObject = myObject;
        }

        public static void main(String args[]) {
                ReassigningReferenceExample testObject1 = new
                                ReassigningReferenceExample("testObject1");
                ReassigningReferenceExample testObject2 = new
                                ReassigningReferenceExample("testObject2");

                // testObject1 now refers to testObject2
                testObject1 = testObject2;

                // Requesting garbage collection
                System.gc();
        }
        @Override
        protected void finalize() throws Throwable {
                // following line will confirm the garbage collected method name
                System.out.println("Garbage collection is succcessful for " +
                                                                this.myObject);
        }
}
```

Output:

Garbage collection is successful for testObject1

Note:

- Since the reference ID of the first object, testObject1, is eventually being used to reference the second object, testObject2, the first object becomes unreachable and is suitable for garbage collection

Example:

```
import java.lang.ref.Cleaner;

public class ReassigningReferenceExample {

	private static String myObject;

	//To obtain a CLEANER instance, call the static create() method on the Cleaner class:
	private static final Cleaner CLEANER = Cleaner.create();

	//Cleanable represents an object and a cleaning action registered in a Cleaner.
	private static Cleaner.Cleanable cleanable;

	//constructor
	public ReassigningReferenceExample(String myObject) {

		this.myObject = myObject;

		//create a Cleanable instance, which will help us register the cleaning actions
		against my object
		//The register() method of a cleaner takes two arguments, the object it is supposed
		// to monitor for cleaning, and the action to perform for cleaning.
	   //cleanable = CLEANER.register(this, () -> System.out.println("Cleaning action "));
		cleanable = CLEANER.register(myObject, new State());
	}

	private static class State implements Runnable {
		public void run() {
			System.out.println("Cleaning action ");
		}
	}

	// close method
	public static void close() {

		//Unregisters the cleanable and invokes the cleaning action.
		//The cleanable's cleaning action is invoked at most once
		//regardless of the number of calls to clean.

		cleanable.clean();
	}

	@Override
	public String toString() {
```

```
        return myObject;
    }

    public static void main(String args[]) {
        ReassigningReferenceExample testObject1 = new
                                        ReassigningReferenceExample("PQR");
        System.out.println(testObject1);

        ReassigningReferenceExample testObject2 = new
                                        ReassigningReferenceExample("STU");
        System.out.println(testObject2);

        // testObject1 now refers to testObject2
        testObject1 = testObject2;

        //Executing close() method
        close();
        System.out.println(testObject1);
        System.out.println(testObject2);

    }
}
```

Output:

```
PQR
STU
Cleaning action
STU
STU
```

- **Nullified the reference**
  - Making all of the variables that reference to it NULL.
  - If the object will have no references ID, hence that object is useless or unreachable and eligible for garbage collection.
  - **Example:** The following is an example code that shows how nullifying the reference variables of an object**.**

```
public class NullifiedReferenceVariablesExample {

        private String myObject;

        public NullifiedReferenceVariablesExample(String myObject) {

            this.myObject = myObject;
```

```
    }

    public static void main(String args[]) {

        NullifiedReferenceVariablesExample testObject1 = new
        NullifiedReferenceVariablesExample("testObject1");

        // Setting testObject1 to Null will qualify it for the garbage
        //collection
        testObject1 = null;

        // Requesting garbage collection
        System.gc();
    }

    @Override
    protected void finalize() throws Throwable {

        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is succcessful for " +
                                                        this.myObject);
    }
}
```

Output:

Garbage collection is successful for testObject1

Note:

- Since there is no longer any reference to testObject1 and its reference variable was made NULL, testObject1 is no longer reachable in the code and becomes suitable for garbage collection. When the garbage collector is called, it finds testObject1 without any reference and removes it from the heap.

Example:

```
import java.lang.ref.Cleaner;

public class NullifiedReferenceVariablesExample {

    private static String myObject;

    //To obtain a CLEANER instance, call the static create() method on the Cleaner class:
    private static final Cleaner CLEANER = Cleaner.create();
```

```
//Cleanable represents an object and a cleaning action registered in a Cleaner.
private static Cleaner.Cleanable cleanable;

//constructor
public NullifiedReferenceVariablesExample(String myObject) {
        this.myObject = myObject;

        //create a Cleanable instance, which will help us register the cleaning actions
        against my object
        //The register() method of a cleaner takes two arguments, the object it is supposed
        to monitor for cleaning, and the action to perform for cleaning.
       //cleanable = CLEANER.register(this, () -> System.out.println("Doing cleaning"));
        cleanable = CLEANER.register(myObject, new State());
}

private static class State implements Runnable {
        public void run() {
                System.out.println("Cleaning action ");
        }
}

// close method
public static void close() {

        //Unregisters the cleanable and invokes the cleaning action.
        //The cleanable's cleaning action is invoked at most once
        //regardless of the number of calls to clean.

        cleanable.clean();
}

@Override
public String toString() {
        return myObject;
}

public static void main(String args[]) {

        NullifiedReferenceVariablesExample testObject1 = new
                                        NullifiedReferenceVariablesExample("EFG");
        System.out.println(testObject1);

        // Setting testObject1 to Null will qualify it for the garbage //collection
        testObject1 = null;
```

```
        //Executing close() method
        close();

        System.out.println(testObject1);
    }
}
```

Output:

EFG
Cleaning action
null

- **Anonymous Objects**
  - Anonymous objects can be used in Java to call methods.
  - In Java, the anonymous objects are different from regular objects.
  - Anonymous objects do not have any reference IDs, but regular objects have reference IDs.
  - As per the criteria, this makes anonymous objects the perfect candidates for garbage collection.
  - **Example:** The following code is an example of a method being used on an anonymous object:

```
public class AnonymousObjectsExample {

    public static void main(String[] args) {

        System.out.println(new
                AnonymousObjectsExample().myMethod());
    }

    public String myMethod() {

        return "I love this book";
    }
}
```

Output:

I love this book

Note:

- The above example demonstrates that anonymous objects can be used to successfully call and run methods.

- **Example:** The following example demonstrates that how garbage collectors can be used to remove anonymous objects from the heap.

```
public class AnonymousObjectsGarbageCollectionExample {

        String myObject;

        public AnonymousObjectsGarbageCollectionExample(String myObject) {

                this.myObject = myObject;
        }

        public static void main(String args[]) {

                // Anonymous Object is being initialized without a reference id
                new AnonymousObjectsGarbageCollectionExample("testObject1");

                // Requesting garbage collector to remove the anonymous object
                System.gc();
        }

        @Override
        protected void finalize() throws Throwable {

                // following line will confirm the garbage collected method name
                System.out.println("Garbage collection is succcessful for " +
                this.myObject);
        }
}
```

Output:

Garbage collection is successful for testObject1

Note:

- Since there is no any reference to the anonymous object, the garbage collector will successfully remove it from the heap.

Example:

```
import java.lang.ref.Cleaner;

public class AnonymousObjectsGarbageCollectionExample {

        private static String myObject;
```

```
//To obtain a CLEANER instance, call the static create() method on the Cleaner class:
private static final Cleaner CLEANER = Cleaner.create();

//Cleanable represents an object and a cleaning action registered in a Cleaner.
private static Cleaner.Cleanable cleanable;

//constructor
public AnonymousObjectsGarbageCollectionExample(String myObject) {

        this.myObject = myObject;
        //create a Cleanable instance, which will help us register the cleaning actions
        against my object
        //The register() method of a cleaner takes two arguments, the object it is supposed
        to monitor for cleaning, and the action to perform for cleaning.
       //cleanable = CLEANER.register(this, () -> System.out.println("Doing cleaning"));
        cleanable = CLEANER.register(myObject, new State());
}

private static class State implements Runnable {
        public void run() {
                System.out.println("Cleaning action on " + myObject);
        }
}

// close method
public static void close() {

        //Unregisters the cleanable and invokes the cleaning action.
        //The cleanable's cleaning action is invoked at most once
        //regardless of the number of calls to clean.

        cleanable.clean();
}

@Override
public String toString() {
        return myObject;
}

public static void main(String args[]) {

        // Anonymous Object is being initialized without a reference id
        new AnonymousObjectsGarbageCollectionExample("I love this book");
```

```
            //Executing close() method
            close();
        }
}
```

Output:

Cleaning action on I love this book