Technische Hochschule Rosenheim

Modul - Fortgeschrittene Programmierkonzepte

Bachelor Informatik

11 - Futures

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Vers. 15.12.2021

Next week

- Register in LC: https://learning-campus.th-rosenheim.de/mod/forum/view.php?id=137001
- Start at 9:45 (**no regular exercise next week!**)

Plan X-Mas break

- Probeklausur verfügbar 23.12 im LC
- Besprechung in der Übung am 13.01.2021 ab 9:45
- Erste reguläre Vorlesung nach den Ferien am 14.01.2021

Agenda for today

Technische Hochschule Rosenheim

What is on the menu for today?

- Short note on GC
- Threads (again)
- Futures, Callable, Executors
- Chaining

Garbage *Collection*

Motivation for GC

Technische Hochschule Rosenheim

- Garbage collection in Java is the process by which Java programs perform automatic memory management.
  - In Java objects are created on the heap, which is a portion of memory dedicated to the program.
  - Unneeded objects needd to be cleaned up.
  - The garbage collector finds these unused objects and deletes them to free up memory.

What is Garbage Collection?

Technische Hochschule Rosenheim

- In C/C++, a programmer is responsible for both the creation and destruction of objects.
- Usually, programmer neglects the destruction of useless objects.
- Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors**.
- **But** in Java, the programmer does not need to care for all objects which are no longer in use.

The **Garbage collector** destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

> The garbage collector is the best example of the Daemon thread as it is always running in the background.

Types of Activities in GC

Technische Hochschule Rosenheim

Two types of garbage collection activity usually happen in Java. These are:

- **Minor or incremental Garbage Collection**: It is said to have occurred when unreachable objects in the young generation heap memory are removed.
- **Major or Full Garbage Collection**: It is said to have occurred when the objects that survived the minor garbage collection and copied into the old generation or permanent generation heap memory are removed. When compared to the young generation, garbage collection happens less frequently in the old generation.

Objects are eligible

Technische Hochschule Rosenheim

1. **Unreachable objects**: An object is said to be unreachable if it doesn't contain any reference to it.

```
Integer i = new Integer(4);
// the new Integer object is reachable  via the reference in 'i'
i = null;
// the Integer object is no longer reachable.
```

1. **Eligibility for garbage collection**: An object is said to be eligible for GC(garbage collection) if it is unreachable. After i = null, integer object 4 in the heap area is suitable for garbage collection in the above image.

Ways to make an object eligible

Technische Hochschule Rosenheim

Even though the programmer is not responsible for destroying useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.

There are generally four ways to make an object eligible for garbage collection:

- Nullifying the reference variable
- Re-assigning the reference variable
- An object created inside the method
- Island of Isolation

Ways to run Garbage Collector

Technische Hochschule Rosenheim

- Once we make an object eligible for garbage collection, it may not destroy immediately by the garbage collector.
- When does JVM run Garbage Collector? - This is not deterministic!
- BUT, We can request JVM to run Garbage Collector. There are two ways to do it :
  - Using `System.gc()` method: System class contain `static method gc()` for requesting JVM to run Garbage Collector.
  - Using `Runtime.getRuntime().gc()` method: Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.
  - The call `System.gc()` is effectively equivalent to the call: `Runtime.getRuntime().gc()`

> There is no guarantee that any of the above two methods will run Garbage Collector.

A final word on `finalize`

Technische Hochschule Rosenheim

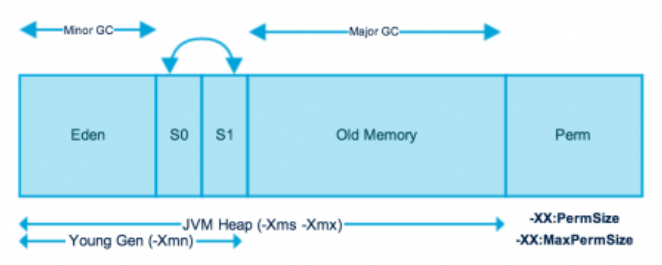
- Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.
- finalize() method is present in Object class with the following prototype.

```
static class Person {

    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}

public static void main(String[] args) {
    Person p = new Person();
    p = null;
    System.gc();
}
```

Java Heap Structure



- The young generation is the place where all the new objects are created. When the young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**.
- Objects that are survived after many cycles of GC, are moved to the Old generation memory space.
- Old Generation memory contains the objects that are long-lived. Old Generation Garbage Collection is called Major GC and usually takes a longer time.

> All the Garbage Collections are “Stop the World” events because all application threads are stopped until the operation completes.

Tools for GC

Technische Hochschule Rosenheim

- `jstat`
  - You can use `jstat` command line tool to monitor the JVM memory and garbage collection activities. It ships with standard JDK, so you don't need to do anything else to get it.
  - For executing jstat you need to know the process id of the application, you can get it easily using ps -eaf | grep java command.
- `jvisualvm`
  - If you want to see memory and GC operations in GUI, then you can use `jvisualvm` tool. Java VisualVM is also part of JDK, so you don't need to download it separately.
  - Just run jvisualvm command in the terminal to launch the Java VisualVM application. Once launched, you need to install Visual GC plugin from `Tools -< Plugins` option.

Futures, *Callable* , Executors ..

Working Asynchronously

Last week, we used threads to do stuff in parallel.

In modern application design, these threads are used to take workload off the main thread so that the actual application (e.g. a GUI or microservice) remains responsive to external input.

Using threads in Java is fairly simple:

1. Extend `Thread` and overwrite `run()`, or create a new thread with reference to a `Runnable`
2. Call `start()` on the thread instance to signal that the thread is ready to run.
3. Optionally use `join()` on the thread instance to wait for its completion.

Working Asynchronously (Code!)

```
Thread t = new Thread(new Runnable() {

    @Override
    public void run() {
        System.out.println("Hello from my custom thread!");
    }

});

t.start();
System.out.println("Hello from the application main thread!")

System.out.println("Waiting for thread to complete...");
t.join();

System.out.println("All done.");
```

Working Asynchronously

From the main application's point of view, the code in `run()` runs *asynchronously*:

- The execution of the `main` method continues right after the `t.start()` call, independent of the execution of `run()`.

On a multi-core computer, the threads may actually be active at the same time, one thread per core.

There are two major drawbacks of this method:

1. There is no (direct) way to provide arguments to or obtaining *results* from the computation done in the thread. This is also reflected in the fact that `run()` does not take any arguments and has return type `void`.
2. There is no (direct) way to communicate potential exceptions that occur in the thread to the main application; the signature of `run()` can not be modified.

Working Asynchronously

A possible solution is to store both arguments and results inside the thread or `Runnable`, and relay possible exceptions to whoever tries to retrieve the result.

```
public class JobWithResult implements Runnable {
    private int a, b, result;
    private Exception e;

    JobWithResult(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        try {
            this.result = a / b;
        } catch (Exception e) {
            this.e = e;
        }
    }

    int getResult() throws Exception {
        if (e != null)
            throw e;
        return result;
    }
}
```

Working Asynchronously

Which can be used in your main program:

```
JobWithResult jwr = new JobWithResult(4, 0);

Thread t = new Thread(jwr);

t.start();

System.out.println("Waiting for thread to complete...");
t.join();

System.out.println("All done.");
try {
    System.out.println(jwr.getResult());
} catch (Exception e) {
    System.out.println("Ooops: " + e.toString());
}
```

The Future of Callables

Technische Hochschule Rosenheim

The code above features three key parts.

1. A `Runnable` which may use external information and stores results as attributes. Possible exceptions raised in the `run()` method are caught and stored as attribute.
2. The main program creates and starts the thread to execute the `run()` method of the above instance.
3. The main program uses the thread's `join()` to wait for completion, and our modified `Runnable`'s `get()` to obtain the result (or raise a possible exception).

Let's refactor the code to separate out the recurring scheme (the mechanics) from the actual business logic (the contents of `run`).

The Future of Callables

Technische Hochschule Rosenheim

Callable

For (1), we'll use a different interface to reflect a return type.

```
interface Callable<V> {
    V call();
}
```

The Future of Callables

Technische Hochschule Rosenheim

Future

For (3), we'll introduce an interface that will allow us to retrieve the *future result* from the `Callable`, once it's done.

```
interface Future<T> {
    T get();
}
```

The Future of Callables

Technische Hochschule Rosenheim

Executor

The remaining part (2) that organizes the thread logistics to run the actual task, we'll stick here:

```
interface Executor {
    <T> Future<T> async(Callable<T> task);
}
```

As you can see, the `Executor` ties things together. Let's see how we can make the threading work.

The Future of Callables

Technische Hochschule Rosenheim

The incoming `Callable` needs to be wrapped into a thread, while watching for possible exceptions. Furthermore, it must return a `Future` that, on `get()`, waits for the thread to finish and then either throws any exception that occurred or returns the result.

```
class SimpleExecutor implements Executor {
    @Override
    public <T> Future<T> async(Callable<T> task) {
        // create anonymous Future instance
        return new Future<T> () {
            Thread t;  // handle on the thread (get needs to wait!)
            T result;  // save the result
            ExecutionException e;  // in case something goes wrong?

            // constructor block
            {
                // create a new thread and start it
                // the runnable "watches" over the task
                t = new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            result = task.call();
                        } catch (Exception ex) {
                            e = new ExecutionException(ex);
                        }
                    }
                });

                t.start();
            }
```

The Future of Callables

Technische Hochschule Rosenheim

...continued

```
            @Override
            public T get() throws InterruptedException, ExecutionException {
                // wait for it...
                t.join();

                // anything fishy?
                if (e != null)
                    throw e;

                // all good, return result!
                return result;
            }
        };
    }
}
```

The Future of Callables

Technische Hochschule Rosenheim

Putting the pieces together, the `SimpleExecutor` simplifies asynchronous execution tremendously:

```
Executor ex = new SimpleExecutor();
int a = 4, b = 0;
Future<Integer> f1 = ex.async(new Callable<Integer>() {
    @Override
    public Integer call() {
        // can use variables from outer scope
        return a / b;
    }
});

// do other things if you like...

try {
    System.out.println(f2.get());
} catch (ExecutionException e) {
    System.out.println("The thread raised an exception: "
        + e.getCause());
}
```

Callables, Futures and Executors since Java 5

Technische Hochschule Rosenheim

Since Java 5, these three parts are realized as Callable (1), Executor (2) and Future (3). The "real" interfaces are slightly different to allow a more faceted use.

```
interface Callable<V> {
    V call();
}
```

```
interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    V get();
    V get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

```
interface ExecutorService extends Executor {
    void execute(Runnable command);  // for convenience
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    <T> Future<T> submit(Runnable task, T result);
    // ...
}
```

Executor

Technische Hochschule Rosenheim

Java 5 also introduced a number of different executors, here are a few examples:

```
Executor executor = Executors.newSingleThreadExecutor();

// executor = Executors.newCachedThreadPool();  // reuses threads
// executor = Executors.newFixedThreadPool(5);  // use 5 threads

executor.execute(new Runnable() {
    public void run() {
        System.out.println("Hello world!");
    }
});
```

Asynchronous on Steroids

Technische Hochschule Rosenheim

The previously introduced `Executor` allows to asynchronously execute a `Callable`, only waiting (blocking) on the executing thread when calling `get()` on the `Future`.

This is convenient enough for basic use cases (e.g. re-query the database after a swipe-down gesture on mobile), but quickly reaches its limits for more complex operations, where the result of one operation gets processed by another.

For the remainder of this chapter, let's consider this rather frequent example: For a cloud-backed application, you write a `displayStatus()` method to

- sign in with username and password; on success you gain a token to
- retrieve the user's status; once received,
- greet the user.

Workflow

Technische Hochschule Rosenheim

Chaining Executors

```
public class Workflow {
    static void displayStatus() throws ExecutionException, InterruptedException {
        final String user = "mustermann";
        final String pass = "12345";  // spaceballs, anyone? :-)

        // log in...
        Future<String> f1 = Executor.async(new Callable<String>() {
            public String call() {
                System.out.println("Authenticating with " + user + ":" + pass);
                return "secrettoken";
            }
        });

        final String token = f1.get();

        // retrieve user
        Future<String> f2 = Executor.async(new Callable<String>() {
            public String call() {
                System.out.println("Retrieving user details with token " + token);
                return "lightly sleep deprived, should get haircut";
            }
        });

        final String details = f2.get();
        System.out.println("Welcome " + user + "! You look " + details);
    }

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        displayStatus();  // blocks until completed!!
    }
}
```

Chaining with CompletableFuture

Technische Hochschule Rosenheim

While this works, the `displayStatus()` method blocks until the status is displayed, thus blocking the remainder of the application from being responsive. Also, reacting to execution exceptions becomes tricky: which parts should be guarded by `try-catch`?

It would be desirable to be able to chain asynchronous actions that depend on each other. The CompletableFuture class added in Java 8 provides exactly that:

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
        // ...
    }
    <U> CompletionStage<U> thenApplyAsync(Function<? super T, ? extends U> fn) {
        // ...
    }
    <U> CompletionStage<U> thenAcceptAsync(Consumer<? super T> action) {
        // ...
    }
    CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn) {
        // ...
    }
    // and much more...
}
```

Chaining with CompletableFuture

Technische Hochschule Rosenheim

The `supplyAsync` accepts a `Supplier<U>`, which is almost identical to `Callable<T>`, but rooted in `java.util.function` for semantic reasons. This method is used to create a `CompletableFuture` that can then be chained with other actions.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
        // ...
    }
...
}
```

Chaining with CompletableFuture

Technische Hochschule Rosenheim

Note the generic typing of `thenApply` and `thenAccept`: Although the `CompletableFuture` is generic in `T`, both methods return `CompletionStage<U>`, i.e. a (possibly) different generic type.

The `thenApply` method takes a `@FunctionalInterface` Function to map values of `? super T` to `? extends U`.

The `thenAccept` method takes a `@FunctionalInterface` Consumer to act on `? super T`.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    <U> CompletionStage<U> thenApplyAsync(Function<? super T, ? extends U> fn) {
        // ...
    }
    <U> CompletionStage<U> thenAcceptAsync(Consumer<? super T> action) {
        // ...
    }
    // and much more...
}
```

Chaining with CompletableFuture

Technische Hochschule Rosenheim

The `exceptionally` method accepts a function that maps a `Throwable` to `? extends T` to handle exceptions gracefully by providing alternate input to the next in line.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn) {
        // ...
    }
    // and much more...
}
```

Since all those are functional interfaces, we can use method references and lambda notation for a cleaner codebase.

Chaining with CompletableFuture

Technische Hochschule Rosenheim

An Example

```
CompletableFuture<?> cf = CompletableFuture.supplyAsync(
    () -> "host:12345")
    .thenApplyAsync(creds -> {
        System.out.println("Authenticating with " + creds);
        return "secrettoken";
    })
    .thenApplyAsync(token -> {
        System.out.println("Retrieving status with token=" + token);
        return "in the mood for holidays";
    })
    .thenAccept(status -> System.out.println(status))
    .exceptionally(ex -> { System.out.println(
        "Oops, something went wrong: " + ex); return null; });

System.out.println("All done!");
```

Chaining with CompletableFuture

Technische Hochschule Rosenheim

You can even combine (synchronize) multiple `CompletableFuture` by using its

```
public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn)
```

Other Concurrency Utilities

Technische Hochschule Rosenheim

Since Java 5, there are a number of other concurrency utilities available.

- `java.util.concurrent` provides thread safe collections
  - `BlockingQueue`: *FIFO* queue that blocks on write-if-full and read-if-empty
  - `ConcurrentMap`: map with atomic read/write operations
- `ThreadLocalRandom` provides random numbers with per-thread randomization
- `AtomicInteger` allows atomic integer increment/decrement
- `CountDownLatch` allows for atomic counting and waiting

CountDownLatch

Technische Hochschule Rosenheim

A frequent use for the `CountDownLatch` is batch processing across multiple threads:

```
List<String> filesToProcess = ...;

CountDownLatch latch = new CountDownLatch(filesToProcess.size());
ExecutorService ex = Executors.newFixedThreadPool(5);

for (String f : filesToProcess) {
    ex.submit(() -> {
        System.out.println("processing file " + f);

        // countdown when done!
        latch.countDown();
    });
}

latch.await();

// shutdown the threads to allow main() to quit.
ex.shutdown();
```

Summary (1)

Technische Hochschule Rosenheim

Use threads to run code asynchronously and in parallel.

- *asynchronously* means the caller/delegator immediately continues execution after queueing the task
- *parallel* means that more than one method is executed at the same time; asynchronous methods are typcially executed in parallel.
- use `join` to wait on threads to complete

Be extra careful when threads *share resources*.

- use thread-safe containers
- protect critical sections with `synchronized`
- minimize blocking time and avoid deadlocks with inter-thread communication (`wait`, `yield`, `notify`, `notifyAll`)

Summary (2)

Technische Hochschule Rosenheim

Use `Future` to retrieve results and handle exceptions within the threads.

- `get()` will block until the thread has completed (and raise possible exception)
- `cancel()` terminates the execution of the task
- `isDone()` returns `true` if the task completed

Use `CompletableFuture` for elegant asynchronous programming.

- use `supplyAsync` to create a chainable `CompletableFuture`
- use `thenApplyAsync` to transform intermediate results
- use `thenAcceptAsyc` to consume final results (end of chain!)
- use `thenCombineAsync` to join multiple `CompletableFuture`

Use `ExecutorServices` when batch-processing large quantities of data, e.g. importing multiple files, scaling images, downloading resources, etc.

Final Thought!