

JAVA / LESSON 35 OF 40

Schedule Tasks

Hand over later execution to the scheduler

Environment: JDK 21, ordinary Paper 1.21.11, client of the same version. Server plugins are loaded by Paper; only explicitly labeled rule tests use main. | 100-120 minutes



60-second activity timer that can be started and stopped

The server needs to continue working and cannot sleep with the countdown. Use tick to schedule delayed and repeated tasks, save task handles, and ensure that repeated starts, manual cancellations, and server stops have clear results.

Prerequisites: Lambda, command and object fields. The new project BlockAcademyL35 continues to use ordinary Paper1.21.11, Java21 and Lesson 31 build scripts.

Before you start: It is recommended to complete Lessons 30-34 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=35&lang=en>

Long code lines may wrap at the > margin marker. The marker is not part of the code.

In this lesson

01. Before starting: first recognize the running position
02. The server uses tick to count beats
03. Do it later. Don't sleep where you are.
04. Repeating tasks requires your own status
05. Only when you get the task ticket can you stop
06. World operations remain on the server main thread
07. Blueprint of Activity Timer
08. The first half of the activity timing program
09. The lower part of the activity timing program
10. Check the beginning and end and start again
11. Expansion challenges and own acceptance records
12. Leave the evidence for the next adventure

01 / 12

Before starting: first recognize the running position

The first 30 lessons lay the foundation for ordinary Java programs. At this stage, the code is connected to the Paper server: the Minecraft client is responsible for entering the world, the server loads the plug-in, and the events and commands hand over player actions to your methods.

This book uses JDK 21 and Paper 1.21.11, and the client version is also consistent. The latest version on the download page may have changed, select it by version number. This book uses normal plugin.yml and the Bukkit API; no claims are made to support Folia at this stage.

The short examples in the picture frame are partial fragments. Read the location description of this section first, and do not save each as a plug-in. The file name will be clearly given for the complete file; when the same file is divided into multiple segments, they will be connected by number, and the package name, import and outer class will only appear once.

- Prediction: Write success conditions and expected messages first.
- Run: After successful construction, install and restart on the dedicated local test server.
- Verification: Ordinary players, management identities and consoles take corresponding paths respectively.

The scene illustrations help to understand the story; the illustrations explain the real rules and are not used as IDE or game actual measurement screenshots. After making Java modifications, rebuild, stop the server, replace it with the new JAR, and start it again; do not treat the performance of the old JAR as the result of the new code.

02 / 12

The server uses tick to count beats

The timekeeper wanted to wait for one second, but he stood sleeping at the door of the server's main thread, and the players behind him lined up in a long queue. The scheduler can make an appointment for a later time, keeping the busy corridor open.

tick is the beat of server work. At normal 20 TPS, 20 ticks is about one second; when there is a lag, the actual time will be longer. Repeating tasks need to save the task handle, and they can be stopped when ending, canceling and deactivating.

The server uses tick to count beats

Element	Explanation
tick	Server working rhythm
20TPS	20 ticks about 1 second
Low TPS	The reality is that the wait will be longer

Milliseconds cannot be used as ticks: 30000 in Lesson 34 is milliseconds; the delay and period of the scheduler are ticks. 20000tick is ideally about 1000 seconds, not 20 seconds.

To clarify the unit and location, see the description in this section

```
long delayTicks = 5L * 20L;  
long durationTicks = 60L * 20L;
```

Checklist

Goal	Number of ticks
3 seconds	60
10 seconds	200
60 seconds	1200

03 / 12

Do it later. Don't sleep where you are.

runTaskLater hands the task to the server and executes it after the specified tick. The current method can end first. Thread.sleep will block the current thread; placing it in the main thread of the server will allow other players to accompany you.

Do it later. Don't sleep where you are.

Step	Explanation
1. Register now	The current method ends immediately
2. Execute later	Check players again after 60 ticks
3. Still online	Send message only if it is not null

The player may have left: Re-search by UUID and check for null when delaying execution. The code example requires importing `java.util.UUID` and `org.bukkit.entity.Player`.

Send a message to online players in three seconds · See the instructions in this section for the location

```
UUID id = player.getUniqueId();
getServer().getScheduler().runTaskLater(this, () -> {
    Player online = getServer().getPlayer(id);
    if (online != null) {
        online.sendMessage(Component.text("Three seconds prompt arrives!"));
    }
}, 60L);
```

Fault experiment: The errors are deliberately kept below and are only observed in the practice copy. Do not merge it into available plugins.

Dangerous main thread wait · Intentional error, predict first and then fix

```
Thread.sleep(3000); // Don't wait like this in a command or sync event
```

04 / 12

Repeating tasks requires your own status

`runTaskTimer` receives first delay and repeat interval. The practice activity is broadcast immediately for 60 seconds, but you have to wait 20 ticks for the first subtraction; if you delay writing 0 and subtract immediately, the opening will be one second shorter.

Repeating tasks requires own status

Step	Explanation
1. Initial value 10	Display or save status first
2. Wait 20 ticks for the first time	Reduce to 9 again
3. Return to zero	The current callback cancels the task

Three positions: this is the plug-in owner; the first 20L is the first delay; the second 20L is the interval for each subsequent time. The example requires `import org.bukkit.scheduler.BukkitRunnable`.

Ten-second countdown experiment · See the instructions in this section for the location

```
new BukkitRunnable() {
    int seconds = 10;
    @Override
    public void run() {
        seconds--;
        getServer().broadcast(Component.text("Remaining " + seconds));
        if (seconds <= 0) cancel();
    }
}.runTaskTimer(this, 20L, 20L);
```

05 / 12

Only when you get the task ticket can you stop

The scheduler returns a `BukkitTask`, which is equivalent to a task ticket. The plug-in stores it in the field `active`. If `active` is null, it means idle; if start is received when there is a ticket, it tells the operator that the timer is already in progress.

Stop and think about it: If you execute the start command when the activity is already running, will there be an extra countdown? How to prove that the task is really canceled after it is stopped?

Only when you get the task ticket can you stop

Element	Explanation
active is empty	OK to start
active is not empty	Refuse to start the second time
Uniform stop	Clear fields after cancel

Figure this out: The implementation should reject repeated starts. If you continue to observe for more than one interval after stopping, there will be no more timing messages; when you start a round again, there should be only one countdown.

Unified cancellation method · See the description in this section for the location

```
private void stopTimer() {  
    if (active != null) active.cancel();  
    active = null;  
}
```

Prevent repeated startup · See the instructions in this section for the location

```
if (active != null) sender.sendMessage("Activity is already timing");  
else startTimer();
```

06 / 12

World operations remain on the server main thread

This lesson all uses synchronous scheduling, and broadcast and world operations are executed on the main thread of the server. Asynchronous tasks are suitable for certain time-consuming calculations or I/O, but many Bukkit world APIs are not accessible from asynchronous threads; asynchrony also introduces contention for shared data.

World operations remain on the server main thread

Element	Explanation
Synchronous short callback	Manipulating Players and the World
Time consuming I/O	Unable to block callbacks per second
Ordinary Paper	This chapter does not claim to support Folia

Don't get the scheduler wrong: Folia uses a different thread and region model. The BukkitScheduler example in this lesson cannot be directly used as a Folia support solution. When the server is stuck, the tick countdown will also slow down.

Keep each callback very short · See the description in this section for the location

```
// Only update one count each time and decide whether to broadcast  
seconds--;  
boolean announce = seconds % 10 == 0 || seconds <= 5;
```

Checklist

work	The processing principles of this lesson
Send activity broadcast	Synchronization tasks
Modify block or player status	Synchronization tasks
Large amounts of file or network I/O	Do not include callbacks per second in this lesson
Ordinary Paper and Folia	This lesson only applies to ordinary Paper

07 / 12

Blueprint of Activity Timer

The challenge provides `/eventtimer start` and `/eventtimer stop`. The console can also be operated, since server-wide activities do not require the operator to have a backpack. By default, only the OP has administrative rights, and ordinary players can only listen to the broadcast.

Blueprint for Activity Timer

Element	Explanation
start	The administrator starts a round
stop	Cancel current task
Natural end	Broadcast once and return to idle

Project path: The main class is located in `src/main/java/de/teacherma/lesson35/EventTimerPlugin.java`. The project name in settings is set to `BlockAcademyL35`, and the build script follows Lesson 31.

Complete plugin.yml · Complete file

```
name: BlockAcademyL35
version: 1.0.0
main: de.teacherma.lesson35.EventTimerPlugin
api-version: '1.21.11'
author: TeacherMa
website: https://www.teacher-ma.de
commands:
  eventtimer:
    description: Manage classroom activity timers
    usage: /eventtimer <start|stop>
permissions:
  blockacademy.timer:
    default: op
```

- Broadcast for 60 seconds at the beginning; then broadcast every 10 seconds, and broadcast every second for the last 5 seconds.
- Repeating start does not create a second task; stop only cancels the current task.
- The natural end is only broadcast once; it can be restarted after the end.

08 / 12

The first half of the activity timing program

The following two pages together are the complete EventTimerPlugin.java. First check the permissions and parameters, and then assign start and stop; the field active saves the common state. Enter eventtimer start on the console without the slash.

Activity timing program first half

Element	Explanation
Entry Check	Permission and one parameter
Start branching	First confirm that there is no activity
Stop branch	Cancel and give truthful feedback

Main class part 1 · This paragraph 1/2, sequential connection

```
package de.teacherma.lesson35;

import java.util.Locale;
import java.util.Objects;
import net.kyori.adventure.text.Component;
import org.bukkit.command.Command;
import org.bukkit.command.CommandSender;
import org.bukkit.plugin.java.JavaPlugin;
import org.bukkit.scheduler.BukkitRunnable;
import org.bukkit.scheduler.BukkitTask;

public final class EventTimerPlugin extends JavaPlugin {
    private BukkitTask active;
    @Override
    public void onEnable() {
        Objects.requireNonNull(getCommand("eventtimer")).setExecutor(this);
    }
    @Override
    public boolean onCommand(CommandSender sender, Command command,
                            String label, String[] args) {
        if (!sender.hasPermission("blockacademy.timer")) {
            sender.sendMessage("Only active administrators can operate the timer");
            return true;
        }
        if (args.length != 1) {
```

Main Class Part 1·This paragraph 2/2, sequential connection

```
        sender.sendMessage("Usage: /eventtimer start or stop");
        return true;
    }
    switch (args[0].toLowerCase(Locale.ROOT)) {
        case "start" -> {
            if (active != null) sender.sendMessage("Activity is already timing");
            else startTimer();
        }
        case "stop" -> {
            if (active == null) sender.sendMessage("There is currently no activity");
            else {
                stopTimer();
                getServer().broadcast(Component.text("Event canceled"));
            }
        }
        default -> sender.sendMessage("Parameters can only be start or stop");
    }
    return true;
}
```

09 / 12

The lower part of the activity timing program

Continue to the previous page. At the natural end, cancel in the callback and then clear active; both manual stop and plug-in deactivation call the same stopTimer. Three ending paths eventually return to idle.

Activity timing program lower half

Step	Explanation
1. 60 to 1	Only one deduction will be made for each callback
2. Return to zero	Broadcast, cancel, clear
3. Disable	Call unified stopTimer

Running rhythm: Under ideal 20TPS, the first callback will be after about 1 second, and the reset will be after about 60 seconds. Only one short calculation is done per second, 60 threads are not created, and sleep is not called.

Main class part 2-sequentially connected to the same source file

```
private void startTimer() {
    getServer().broadcast(Component.text("Activity starts, remaining 60 seconds"));
    active = new BukkitRunnable() {
        private int seconds = 60;
        @Override
        public void run() {
            seconds--;
            if (seconds <= 0) {
                getServer().broadcast(Component.text("Time is up, the activity is over!"));
                cancel();
                active = null;
                return;
            }
            if (seconds % 10 == 0 || seconds <= 5) {
                getServer().broadcast(Component.text("Remaining " + seconds + " seconds"));
            }
        }
    }.runTaskTimer(this, 20L, 20L);
}
private void stopTimer() {
    if (active != null) active.cancel();
    active = null;
}
@Override
public void onDisable() { stopTimer(); }
}
```

Check the beginning and end and start again

The timer is most likely to make mistakes at the boundary: seconds are deducted as soon as it is started, broadcast is finished twice, and it is still talking after stopping. Use the following table to check item by item and match the order in the log with the messages in the game.

Check start end and start again

Element	Explanation
Beginning	Cannot lose one second immediately
End	Cannot broadcast repeatedly
Start again	Only one new countdown

Real time comparison: You can use a stopwatch to observe, but do not judge the slowdown under low TPS as a subtraction formula error. If the activity must be deadlined strictly according to the actual time, the deadline should be recalculated with each callback.

Checklist

Test	Expected
Start again immediately after start	There is only one countdown
stop during operation	Cancel the message once, then be quiet
stop when idle	Prompt that there is currently no activity
Let 60 seconds end completely	50, 40, 30, 20, 10, 5 to 1, end
Start again after the end	Start again from 60 seconds
Stop server during timing	Plug-in cleaning task
Ordinary players start	Permission prompt, no task

Expansion challenges and own acceptance records

Leave the source code, actual running results and a problem record. Explain it in your own words first, and then run the verification; where the results are different from the predictions, they are often the most worthy of discussion.

Expansion tasks and clearance review

Element	Explanation
Wooden Pickaxe	Delay prompts take into account offline
Iron Pickaxe	Explain the 10 second countdown
Diamond Pickaxe	Day and night reminders can be started and stopped

Next step: Lesson 36 continues to turn server rules into configurable functions.

Official extended reading: docs.papermc.io/paper/dev/scheduler/

- Wooden Pickaxe: Use `runTaskLater` to prompt the player after 3 seconds, and no error will be reported when the player exits.
- Iron Pickaxe: Make a 10-second countdown and explain where the first delay, interval, and end conditions are respectively.
- Diamond Pickaxe: Make a day and night reminder that can be started and stopped to ensure that repeated starts do not overlap and are canceled when disabled.
- Acceptance requirements: record six moments: start, remaining 1, end, manual cancellation, restart and stop. Make it clear that this lesson uses tick timing.

Checklist

I can complete it independently	Mastered	Practice again
Can distinguish milliseconds and ticks	☐	☐
Will save and cancel the task	☐	☐
Understand the boundaries of synchronization tasks	☐	☐

12 / 12

Leave the evidence for the next adventure

Divide "can compile", "number rules pass" and "actual performance of the game is correct" into three columns. Scenarios that are not running are written to be tested; when bugs are discovered, trigger conditions, expected, actual, minimum repairs, and retest results are recorded.

Your verification record

Category	The operation you selected	Evidence left behind
Normal	A successful path for this lesson	Version and actual output
Border	The moment before the end, the maximum value or the full backpack	Compare item by item with contract
Reject	Wrong parameters, no permissions or irrelevant events	The feedback is clear and the status has not been changed by mistake.

- Explain why each arrow in this lesson points from here to there; the parallel cards have no sequential meaning.
- First change a rule, then modify the description and expectations at the same time, and retest with the same input.

Technical check: This lesson is based on Paper 1.21.11 API and Java SE 21. The reference document may be switched to a newer version, so pay attention to the version number when checking it.

<https://jd.papermc.io/paper/1.21.11/>

<https://docs.papermc.io/paper/dev/>

<https://docs.oracle.com/en/java/javase/21/docs/api/>