

JAVA / LESSON 34 OF 40

Data for Every Player

Keep separate cooldowns for each player with UUID

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



Mutual /dash cooling

Everyone shares a Map, and each player has his own UUID tag. Write the cooldown as a deadline, verify the boundaries with a controllable test clock, and then receive the real sprint command.

Front-end backpack: HashMap, commands, permissions and simple arithmetic. The new project BlockAcademyL34 follows the build script from Lesson 31, using Java21 and Paper1.21.11.

Before you start: It is recommended to complete Lessons 26, 31-33 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=34&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 player's name will change and the identity key must be stable
03. Each player has his own grid
04. Record the moment when you can act again
05. It's not over yet with one millisecond left.
06. Cooling rules are independent into a class
07. Let the fake clock wait for you for thirty seconds
08. The sprint direction is an arrow
09. Only start cooling after successful execution
10. Exiting the game does not mean that the cooldown disappears
11. Two Java files are required for the sprint project
12. Sprint the first half of the main class
13. sprint main class lower half
14. It takes two players to reveal shared cooldown
15. Expansion challenges and own acceptance records
16. Rule verification that can run independently
17. Leave the evidence for the next adventure

01 / 17

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 / 17

The player's name will change and the identity key must be stable

Two players took turns to sprint, and the second player was told "You are still cooling down" as soon as he started. It turns out that the program puts the whole village into the same stopwatch. Each player uses his or her own UUID to find his or her own record, so that there will be no interruptions during cooling.

The key of the Map is the player's identity, and the value is the next time the action is allowed. Just compare the current time with this moment. There is no need to create tasks for everyone every second. By passing the time as a parameter during testing, you don't have to wait for thirty seconds.

The player's name will change and the identity key must be stable

Element	Explanation
UUID	Identity key provided by server
Display name	For readers and players to see
Data recording	Press the button to find your own state

Short name and long number: the name is convenient for people to read, and the UUID is convenient for programs to associate data. This lesson directly uses the UUID provided by the server, instead of spelling one out using the player's name.

Read the identity key · See the description in this section for the location

```
UUID id = player.getUniqueId();  
String displayName = player.getName();
```

Appropriate field type · The location is explained in this section

```
private final Map<UUID, Integer> mined = new HashMap<>();
```

03 / 17

Each player has his own grid

If only one global frequency variable is written, and Alex digs once, it will also increase when Steve checks it. Map uses UUID to separate records: everyone shares a box, and each person has his own label.

Each player has his own grid

Element	Explanation
Alex's record	Only update Alex
Steve's record	Do not follow the increase
Missing records	getOrDefault provides initial value

Online players are not data files: long-term Map stores UUID and ordinary data to avoid long-term storage of Player objects. When you need to operate an online player, use the UUID to find and check whether they are online.

Cumulative mining times · See the description in this section for the location

```
UUID id = player.getUniqueId();
mined.merge(id, 1, Integer::sum);
int count = mined.getDefault(id, 0);
```

New players default to zero. See the description in this section for the position.

```
UUID other = UUID.randomUUID(); // Only for local practice
System.out.println(mined.getDefault(other, 0));
```

Output

0

04 / 17

Record the moment when you can act again

Assume that the current time is 1000 milliseconds and the cooling lasts for 30000 milliseconds, then the time that can be used again is 31000. Each player saves his own readyAt and compares the current time to know whether it is still cooling down.

Stop and think about it: It is still 1 millisecond away from the end of the cooling, should it be allowed? If the record is deleted directly when the player exits, what kind of bypass will it bring?

Record the moment when you can act again

Element	Explanation
Now 1000ms	Start 30 seconds cooling
30999ms before end	1ms left, reject
Ending 31000ms	0 remaining, allowed

Think this through: reject before the time of permission comes. Deleting after exiting will cause reconnection to skip cooling, so the cleanup strategy should retain unfinished records.

Time model: This example uses the system time as the basis for short-term cooling; adjusting the system clock may change the displayed remaining time. System.nanoTime can be studied with the advanced function of measuring only the elapsed time, but its starting point and unit are different and cannot be directly mixed with millisecond archives.

Calculate the deadline · See the description in this section for the location

```
long now = System.currentTimeMillis();
long readyAt = now + 30_000L;
boolean cooling = now < readyAt;
```

Unified time unit · See the description in this section for the location

```
long durationMs = 30_000L; // 30 seconds, no 30 milliseconds
long threeSeconds = 3L * 1000L;
```

05 / 17

It's not over yet with one millisecond left.

Players don't like seeing "wait another 0 seconds" but still not being able to act. Milliseconds are used to determine the cooldown, and are rounded up when displayed to the player. As long as there is 1 millisecond left, the interface will be displayed for 1 second.

It's not over yet with even one millisecond left.

Element	Explanation
Logical judgment	Use remaining milliseconds
Player display	Round up to seconds
Boundary evidence	1ms also displays 1 second

Predict by hand: cover the right column first, and then correct the answer after calculating. Direct division by 1000 will truncate the decimals, which is the source of "0 seconds but it cannot be used".

The remaining time is not less than zero · See the description in this section for the position

```
long remainingMs = Math.max(0L, readyAt - now);
```

The integer is rounded up. See the description in this section for the position.

```
long seconds = (remainingMs + 999L) / 1000L;
```

Checklist

Milliseconds remaining	Display seconds
0	0
1	1
999	1
1000	1
1001	2

06 / 17

Cooling rules are independent into a class

Create new `src/main/java/de/teacherma/lesson34/Cooldowns.java`. The method receives `now` and does not secretly read the clock internally, so that the time can be directly pushed to the boundary during testing.

Cooling rules are separated into a class

Element	Explanation
remaining	Calculate remaining time
start	Record deadline
prune and clear	Clean up records by policy

Parameter agreement: `id` uses the real player UUID, `durationMs` is a short-term non-negative number of milliseconds, and `now` remains the same in one operation. This class is used for short-term cooling of exercises on the main thread and does not persist across restarts.

Complete Cooldowns class · Complete file

```
package de.teacherma.lesson34;

import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

public final class Cooldowns {
    private final Map<UUID, Long> readyAt = new HashMap<>();
    public long remaining(UUID id, long now) {
        return Math.max(0L, readyAt.getOrDefault(id, now) - now);
    }
    public void start(UUID id, long now, long durationMs) {
        if (durationMs < 0) throw new IllegalArgumentException("Cooling cannot be negative");
        readyAt.put(id, now + durationMs);
    }
    public long seconds(UUID id, long now) {
        long ms = remaining(id, now);
        return (ms + 999L) / 1000L;
    }
    public void prune(long now) {
        readyAt.entrySet().removeIf(entry -> entry.getValue() <= now);
    }
    public void clear() { readyAt.clear(); }
}
```

07 / 17

Let the fake clock wait for you for thirty seconds

You don't have to actually stand in front of the screen for a few seconds. Create a new ordinary test class and call the following code in main. All numbers are test clocks; this makes boundary testing fast and easy to reproduce.

Let the fake clock wait thirty seconds for you

Element	Explanation
Control test time	No need to really wait 30 seconds
Two UUIDs	Prove state independence
Two boundaries	1ms before end and exactly end

Key evidence: The last line is Steve. Alex is still cooling down, but Steve can move, proving that the status is separated by player. Retesting negative cooling parameters should throw an exception.

Boundary test of two players · See the description in this section for the position

```
Cooldowns c = new Cooldowns();
UUID alex = UUID.randomUUID();
UUID steve = UUID.randomUUID();
c.start(alex, 1000L, 30_000L);
System.out.println(c.remaining(alex, 1000L));
System.out.println(c.remaining(alex, 30999L));
System.out.println(c.seconds(alex, 30999L));
System.out.println(c.remaining(alex, 31000L));
System.out.println(c.remaining(steve, 2000L));
```

Output

```
30000
1
1
0
0
```

08 / 17

The sprint direction is an arrow

getDirection returns the player's sight direction. Set Y to 0 to get a horizontal arrow, normalize and multiply by 0.9, and finally give a slight upward velocity of 0.2. When the player looks at the sky almost vertically, the horizontal arrow may be close to zero, so reject it first.

The sprint direction is an arrow

Step	Explanation
1. Horizontal component	First set Y to 0
2. Length check	Close to 0 and reject first
3. Submission speed	After normalization, horizontal and upward components are given

Before entering the next step along the arrow, the conditions of the previous step must be met; the rejected branch returns early.

Speed is not distance: this is not a teleport fixed at 0.9 blocks. Terrain, collision, resistance, and server rules affect actual movement. Try it on a flat testing area with space next to it.

Prevent normalized zero vector · See the description in this section for the location

```
Vector direction = player.getLocation().getDirection().setY(0);
if (direction.lengthSquared() < 0.0001) {
    player.sendMessage(Component.text("Please look straight ahead before sprinting"));
    return true;
}
```

Give the player a speed and position. See the instructions in this section.

```
player.setVelocity(direction.normalize().multiply(0.9).setY(0.2));
```

09 / 17

Only start cooling after successful execution

If the buckle is cooled too early, the player will have to wait 30 seconds if he raises his head and makes a mistake once, just like a vending machine swallows money first and then checks whether there is a drink. Verify in sequence and record the cooling only after setVelocity is actually called.

Only start cooling after successful execution

Step	Explanation
1. Legal request	Type, authority, parameters pass
2. Submission speed	Only execute if the direction is valid
3. Record cooling	Start again at the end of the successful path

Before entering the next step along the arrow, the conditions of the previous step must be met; the rejected branch returns early.

The meaning of success: This means that the speed is checked and submitted through this plug-in, and the final moving distance is not guaranteed. Other plugins may intercept or modify speed; complex skills require further definition of success rules.

Use the same current time. See the instructions in this section for the location.

```
long now = System.currentTimeMillis();
if (cooldowns.remaining(id, now) > 0) {
    player.sendMessage(Component.text(
        "Wait again " + cooldowns.seconds(id, now) + " seconds"));
    return true;
}
```

End of successful path · See the description in this section for the location

```
player.setVelocity(direction.normalize().multiply(0.9).setY(0.2));
cooldowns.start(id, now, 30_000L);
```

10 / 17

Exiting the game does not mean that the cooldown disappears

If a player deletes his cooldown as soon as he logs out, he will be able to sprint again immediately after logging in again. This lesson retains unexpired records, and clears expired items the next time someone executes a command; it will be cleared when the plug-in is deactivated.

Exiting the game does not mean that the cooldown disappears

Element	Explanation
Exit and re-enter	Unfinished record retention
Record expired	Clean up when someone executes the command
Server restart	Memory cooling and clearing in this chapter

Small class size: prune will scan the current records to suit the number of people practicing. Large servers can be cleaned in batches at regular intervals; learn from the 35th expansion and redesign. Memory Map is not a permanent archive.

Lazy cleaning · See the description in this section for the location

```
long now = System.currentTimeMillis();
cooldowns.prune(now);
```

Cleared when deactivated · See the description in this section for the location

```
@Override
public void onDisable() {
    cooldowns.clear();
}
```

Checklist

Action	Behavior of this lesson
Player exits and re-enters	Unexpired cooling continues
Plug-in deactivation or server restart	Clear memory data
Someone executes the command after the record expires	Delete expired items

11 / 17

Two Java files are required for the sprint project

src/main/java/de/teacherma/lesson34 decentralized Cooldowns.java and DashPlugin.java. They are in the same package, so the main class can use Cooldowns directly. The resource file only declares the main class and does not need to register ordinary auxiliary classes.

A sprint project requires two Java files

Element	Explanation
Cooldowns.java	Just numbers and time
DashPlugin.java	Connect command and player
plugin.yml	Only register the plug-in main class

Prepare to build: The project name in settings is set to BlockAcademyL34; build.gradle.kts follows Lesson 31. Don't stick two public classes into the same Java file.

Complete plugin.yml · Complete file

```
name: BlockAcademyL34
version: 1.0.0
main: de.teacherma.lesson34.DashPlugin
api-version: '1.21.11'
author: TeacherMa
website: https://www.teacher-ma.de
commands:
  dash:
    description: Sprint forward
    usage: /dash
permissions:
  blockacademy.dash:
    default: true
```

12 / 17

Sprint the first half of the main class

Create a new DashPlugin.java, copy this page first, and then continue to the next page. The main class holds a Cooldowns object, which internally stores records for each player by UUID.

Sprint the first half of the main class

Element	Explanation
A plug-in field	Hold a copy of Cooldowns
Map multiple keys	One record per player
A request	Query and update use the same now

The field is not one for each person: the cooldowns field belongs to the plug-in; the key value record in the Map belongs to each player. There is no need to create a new plug-in for each person, nor to use a global unified countdown.

DashPlugin Part 1 · Sequentially connected to the same source file

```
package de.teacherma.lesson34;

import java.util.Objects;
import net.kyori.adventure.text.Component;
import org.bukkit.command.Command;
import org.bukkit.command.CommandSender;
import org.bukkit.entity.Player;
import org.bukkit.plugin.java.JavaPlugin;
import org.bukkit.util.Vector;

public final class DashPlugin extends JavaPlugin {
    private final Cooldowns cooldowns = new Cooldowns();
    @Override
    public void onEnable() {
        Objects.requireNonNull(getCommand("dash")).setExecutor(this);
    }
    @Override
    public boolean onCommand(CommandSender sender, Command command,
                             String label, String[] args) {
        if (!(sender instanceof Player player)) {
            sender.sendMessage(Component.text("Sprinting requires an in-game body"));
            return true;
        }
        if (!player.hasPermission("blockacademy.dash")) {
            player.sendMessage(Component.text("You do not have sprint permission yet"));
            return true;
        }
        if (args.length != 0) {
            player.sendMessage(Component.text("Usage: /dash"));
            return true;
        }
    }
}
```

13 / 17

sprint main class lower half

Continue to the previous page. Querying, prompting, direction validation, execution, and logging all happen in a single command invocation. Branches that fail the check are returned early to facilitate reading from top to bottom.

Sprint the lower half of the main class

Element	Explanation
Query	The remaining time determines the branch
Direction	Zero vector does not execute
Action completed	Record cooling and feedback

Start game acceptance: build and restart the test server. Execute /dash, and then execute it again immediately. The second time should only display the remaining seconds; wait 30 seconds before dashing again.

DashPlugin Part 2 · Sequentially connected to the same source file

```
        long now = System.currentTimeMillis();
        cooldowns.prune(now);
        var id = player.getId();
        if (cooldowns.remaining(id, now) > 0) {
            player.sendMessage(Component.text(
                "Wait again " + cooldowns.seconds(id, now) + " seconds"));
            return true;
        }
        Vector direction = player.getLocation().getDirection().setY(0);
        if (direction.lengthSquared() < 0.0001) {
            player.sendMessage(Component.text("Please look straight ahead before sprinting"));
            return true;
        }
        player.setVelocity(direction.normalize().multiply(0.9).setY(0.2));
        cooldowns.start(id, now, 30_000L);
        player.sendMessage(Component.text("Sprint! Need to wait next time 30 seconds"));
        return true;
    }
    @Override
    public void onDisable() { cooldowns.clear(); }
}
```

14 / 17

It takes two players to reveal shared cooldown

Single-person testing cannot see the error of "everyone shares the same deadline". Ask two students to connect to the test server respectively, go through the following table, and then check the exit and re-entry and time boundaries.

It takes two players to reveal shared cooldown

Element	Explanation
Alex uses it first	Alex enters cooling
Steve will use it immediately	Steve is still available
Alex reuse	Only received the remaining time prompt

Your checklist: Have students explain the units of readyAt, now, and remaining, and then explain why records cannot be cleared by exiting. Write "when data disappears" into the function description.

Checklist

Operation	Expected
Steve uses it immediately after Alex uses it	Both succeeded
Alex uses it again immediately	Prompt remaining seconds
Exit during cooling and re-enter immediately	Still limited by remaining cooling
Use after viewing the sky almost vertically	Prompt to look up, no new cooling
With extra parameters /dash now	Display usage, no new cooling
Console input dash	Clear prompt, no type conversion error
Use after stopping and restarting	Memory cooling cleared

15 / 17

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	Independently record mining numbers
Iron Pickaxe	Test 1ms and 0ms
Diamond Pickaxe	Player and skill two layers of independent keys

Next stop: Lesson 35: Use the scheduler to automatically update the countdown.
Official extended reading: jd.papermc.io/paper/1.21.11/org/bukkit/entity/Player.html

- Wooden pickaxe: Use Map to record the number of times each player is allowed to mine, and players who have not appeared before will display 0.
- Iron pick: Set the cooling to 10 seconds, verify that the remaining 1 millisecond displays 1 second, and displays 0 seconds when it expires.
- Diamond Pickaxe: Each has independent cooling for sprint, treatment, and tool kit. It is designed with two layers of keys for players and skills, and it is tested that they do not affect each other.
- Acceptance requirements: Use two UUIDs and controllable test time, verify that they do not affect each other, 1 millisecond before the end, exactly end, and rejoin after exiting.

Checklist

I can complete it independently	Mastered	Practice again
Use UUID to associate player records	☐	☐
Can calculate the remaining time and round it up	☐	☐
Can explain exit and restart strategies	☐	☐

16 / 17

Rule verification that can run independently

Create new checks/L34RuleCheck.java in the project and save the following code completely. Create a new checks/sources.txt, write the following two paths, and then execute the command from the project root directory. This is a normal Java test, not the main plugin class.

L34RuleCheck.java · Complete file

```
import de.teacherma.lesson34.Cooldowns;
import java.util.UUID;

public class L34RuleCheck {
    public static void main(String[] args) {
        Cooldowns c = new Cooldowns();
        UUID alex = new UUID(0, 1), steve = new UUID(0, 2);
        c.start(alex, 1000L, 30_000L);
        System.out.println(c.remaining(alex, 1000L));
        System.out.println(c.remaining(alex, 30999L));
        System.out.println(c.seconds(alex, 30999L));
        System.out.println(c.remaining(alex, 31000L));
        System.out.println(c.remaining(steve, 2000L));
    }
}
```

Output

```
30000
1
1
0
0
```

checks/sources.txt · Full content

```
src/main/java/de/teacherma/lesson34/Cooldowns.java
checks/L34RuleCheck.java
```

Project root directory terminal command · Execute two lines respectively

```
javac -encoding UTF-8 -d checks/out @checks/sources.txt
java -cp checks/out L34RuleCheck
```

Pure rules only prove the digital contract, and permissions, player actions, event registration and server loading are still checked according to the previous game acceptance form.

17 / 17

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/>