Bazel in Java explained

give me the deep and detailed explaination of Bazel in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Alright Piyush — since you like **deep, crystal-clear explanations in simple language**, I'll explain **Bazel for Java** from **absolute basics → internals → real Java usage → comparisons → mental model**, so that **no confusion remains**.

Take your time with this — Bazel is powerful but conceptually different from Maven/Gradle.

1️⃣ What is Bazel? (Big Picture)

Bazel is a build system.

A **build system** is responsible for:

- Compiling source code
- Managing dependencies
- Running tests
- Packaging artifacts (JARs, WARs, binaries)
- Ensuring builds are **correct, fast, and repeatable**

👉 Examples of build systems:

- Maven
- Gradle
- Ant
- **Bazel**

Why Bazel Exists

Bazel was created by **Google** to solve problems that appear when:

- Projects become **very large**
- Multiple languages are used together
- Build time becomes very slow
- Builds behave differently on different machines

Google uses Bazel to build **millions of lines of code**, including Java.

2️⃣ Core Philosophy of Bazel (VERY IMPORTANT)

Bazel is built on **3 fundamental principles**:

1. Hermetic Builds

> A build depends ONLY on declared inputs, nothing else.

- Same input → same output
- No hidden dependencies
- No "works on my machine" problem

2. Reproducibility

> The build result is identical on every machine.

If your code builds on:

- Your laptop
- CI server
- Teammate's system

➡️ The output will be **bit-for-bit identical**

3. Incrementality

> Bazel only rebuilds what actually changed.

If you change **1 Java file**, Bazel:

- Rebuilds only affected targets
- Reuses cached results for everything else

This makes Bazel **extremely fast** for large projects.

3️⃣ Bazel vs Maven/Gradle (Mindset Shift)

Concept	Maven / Gradle	Bazel
Build file	`pom.xml`, `build.gradle`	`BUILD`
Dependency resolution	Automatic & transitive	Explicit & strict
Incremental build	Limited	Very strong
Cache	Local	Local + Remote
Large monorepo	Hard	Excellent

Concept	Maven / Gradle	Bazel
Learning curve	Easy	Steep

⚠️ **Important difference**
Maven/Gradle focus on **projects**
Bazel focuses on **targets**

4️⃣ Bazel Terminology (Must Know)

Before touching Java, you MUST understand these words.

---

1. Workspace

A **workspace** is the **root directory** of a Bazel project.

It contains:

```
css

WORKSPACE
BUILD
src/
```

WORKSPACE file

- Defines **external dependencies**
- Defines repository rules
- Similar to Maven's `<repositories>`

Example:

```
text

WORKSPACE
```

Even an empty `WORKSPACE` file makes the folder a Bazel workspace.

---

2. BUILD File

A `BUILD` file tells Bazel:

> "How should this code be built?"

Every directory can have a `BUILD` file.

Example:

```
css

src/main/java/com/app/BUILD
```

---

3. Target (MOST IMPORTANT)

A **target** is something Bazel can build.

Examples:

- A Java library
- A Java binary
- A test
- A generated file

Each target has:

- A **name**
- A **rule**
- Inputs
- Outputs

You always run Bazel like this:

```
bash

bazel build //path:target_name
```

---

4. Rule

A **rule** defines:

> How to turn inputs into outputs

Examples:

- `java_library`
- `java_binary`
- `java_test`

Rules are written in **Starlark** (Python-like language).

5️⃣ Java Support in Bazel

Bazel provides **native Java rules**.

Common Java rules:

Rule	Purpose
`java_library`	Compile reusable Java code
`java_binary`	Create an executable Java app
`java_test`	Run JUnit tests
`java_import`	Import prebuilt JARs

6️⃣ Minimal Java Project Using Bazel

Let's build a **simple Java app** step by step.

🗂️ Project Structure

```
css

bazel-java-demo/
│
├── WORKSPACE
├── BUILD
└── src/
    └── Main.java
```

📄 Main.java

```
java

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello from Bazel!");
    }
}
```

📄 BUILD File

```
python

java_binary(
    name = "hello_app",
    srcs = ["src/Main.java"],
    main_class = "Main",
)
```

Explanation (Line by Line)

- `java_binary` → rule type

- `name` → target name
- `srcs` → source files
- `main_class` → entry point

▶️ Build and Run

```
bash

bazel build //:hello_app
bazel run //:hello_app
```

✔️ Output:

```
csharp

Hello from Bazel!
```

7️⃣ `java_library` (Very Important for Real Projects)

In real projects:

- You don't put everything in one binary
- You split code into libraries

Example Structure

```
css

src/
 ├── util/
 │    ├── Util.java
 │    └── BUILD
 └── app/
      ├── Main.java
      └── BUILD
```

---

util/BUILD

```
python

java_library(
    name = "util_lib",
    srcs = ["Util.java"],
)
```

---

app/BUILD

```
python

java_binary(
    name = "app",
    srcs = ["Main.java"],
    deps = ["//src/util:util_lib"],
    main_class = "Main",
)
```

Key Concept: `deps`

- **deps must be explicitly declared**
- If you forget → build fails
- No hidden dependencies allowed

This is called **strict dependency checking**.

8️⃣ Dependency Management in Bazel (Java)

Unlike Maven:

- Bazel does NOT auto-download dependencies magically

You must:

1. Declare dependencies in `WORKSPACE`
2. Reference them in `BUILD`

Using Maven Dependencies

Bazel provides `rules_jvm_external`.

Example (simplified):

```
python

maven_install(
    artifacts = [
        "org.springframework:spring-core:6.1.0",
    ],
    repositories = [
        "https://repo1.maven.org/maven2",
    ],
)
```

Then use in BUILD:

```
python

deps = ["@maven//:org_springframework_spring_core"]
```

👉 This explicitness is why Bazel builds are **reliable**.

9️⃣ Testing in Bazel (JUnit)

```
python

java_test(
    name = "util_test",
    srcs = ["UtilTest.java"],
    deps = [
        "//src/util:util_lib",
        "@maven//:junit_junit",
    ],
)
```

Run:

```
bash

bazel test //src/util:util_test
```

🔟 Caching & Performance (Why Bazel is FAST)

Local Cache

- If input hasn't changed → result reused

Remote Cache

- Team shares build results
- CI builds once, developers reuse

This is why Bazel is perfect for:

- Monorepos
- Large teams
- Microservices + shared libraries

1️⃣1️⃣ Why Bazel Feels Hard Initially

Because Bazel:

- Forces correctness
- Forces explicit dependencies
- Rejects shortcuts

But this gives:
✔️ Faster builds

✔️ No flaky behavior
✔️ Scalable architecture

1️⃣2️⃣ When Should YOU Use Bazel?

Use Bazel if:

- Very large Java codebase
- Monorepo
- Multiple languages (Java + Go + Python)
- CI speed is critical

Don't use Bazel if:

- Small Spring Boot app
- Simple REST service
- Solo developer project

For most **Spring Boot backend projects**, **Gradle is enough**.
Bazel shines at **Google-scale systems**.

1️⃣3️⃣ Mental Model (Remember This)

> **Bazel = Graph of Targets**

- Nodes → targets
- Edges → dependencies
- Change one node → rebuild only affected nodes

If you understand this graph idea → Bazel becomes clear.

1️⃣4️⃣ Final Summary

- Bazel is a **powerful, strict, scalable build system**
- Java support is **first-class**
- Explicit dependencies = correctness
- Harder to learn, easier to scale
- Best for **large, long-term projects**

If you want next:

- 🔥 **Bazel vs Maven vs Gradle (real Java examples)**

- 🔥 **Bazel + Spring Boot**
- 🔥 **Monorepo design using Bazel**
- 🔥 **Bazel interview questions**

Just tell me 👍