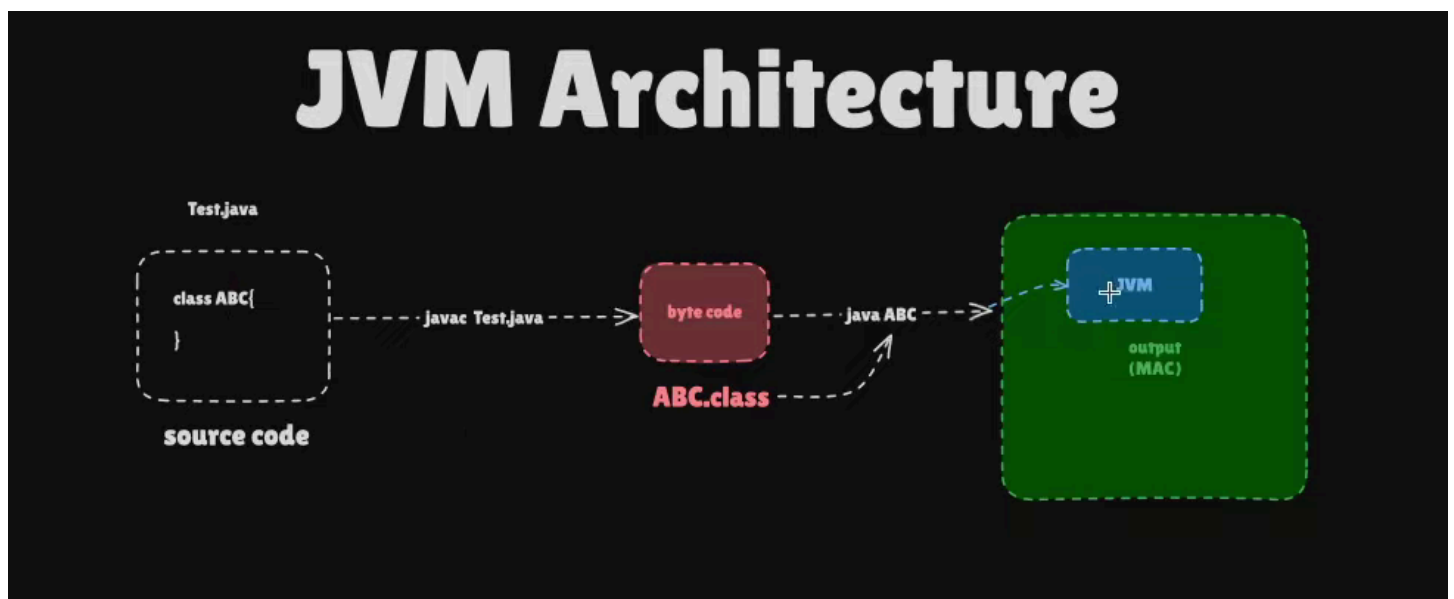


JAVA Internal Working (JVM)

Here is a clear English summary of the video transcript on **JVM Architecture** (Java Virtual Machine), covering how Java programs run and the internal structure of the JVM (with emphasis on changes from Java 8 onward):

How a Java Program Runs (Basic Flow)

1. You write **source code** (.java file) in a high-level language.
2. The **javac** compiler compiles it into **bytecode** (.class file) – platform-independent intermediate code that humans can't easily read, but the JVM understands.
3. To run the program: `java ClassName` → The JVM (part of JRE – Java Runtime Environment) loads and executes this bytecode.
4. The JVM provides a virtual/abstract machine that makes Java "write once, run anywhere" by handling platform-specific details.

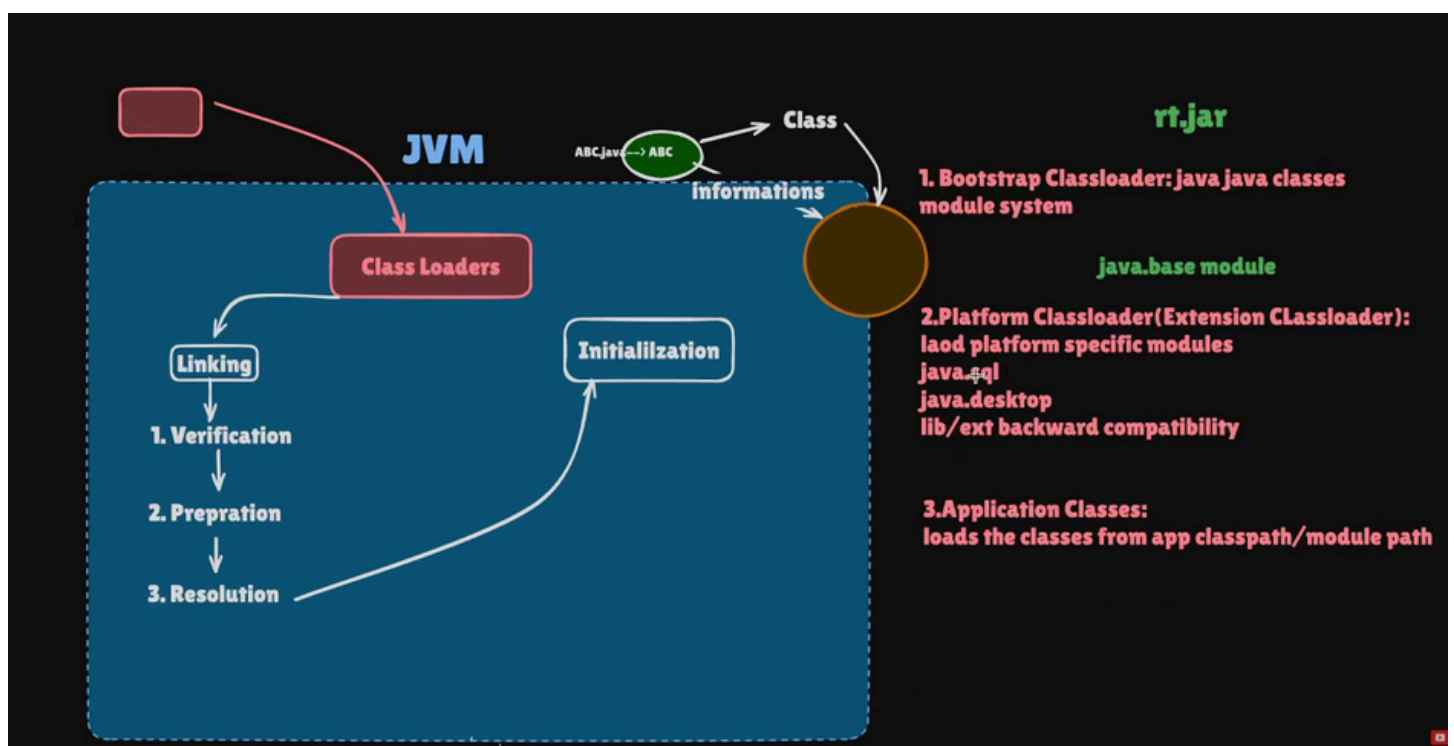


Main Components of JVM Architecture

The JVM has three primary subsystems:

1. **Class Loader Subsystem**

- Loads .class files into memory.
- Three main types of class loaders (with changes in Java 9+):
 - **Bootstrap ClassLoader** (primordial, native code) → Highest priority. Loads core Java classes (from **java.base** module in Java 9+; earlier from rt.jar).
 - **Platform ClassLoader** (Java 9+) → Replaced the old **Extension ClassLoader**. Loads platform-specific modules (e.g., java.desktop) and maintains backward compatibility (loads from ext/ folder if needed).
 - **Application ClassLoader** (also called System ClassLoader) → Loads your application's classes from the classpath or module path.
- Loading process includes:
 - **Loading** → Read bytecode into memory.
 - **Linking**:
 - Verification → Check bytecode is valid and follows JVM specs (security).
 - Preparation → Allocate memory for static variables and set default values.
 - Resolution → Convert symbolic references (e.g., method/class names) to direct references.
 - **Initialization** → Execute static blocks, assign static variable values (static initialization happens here).

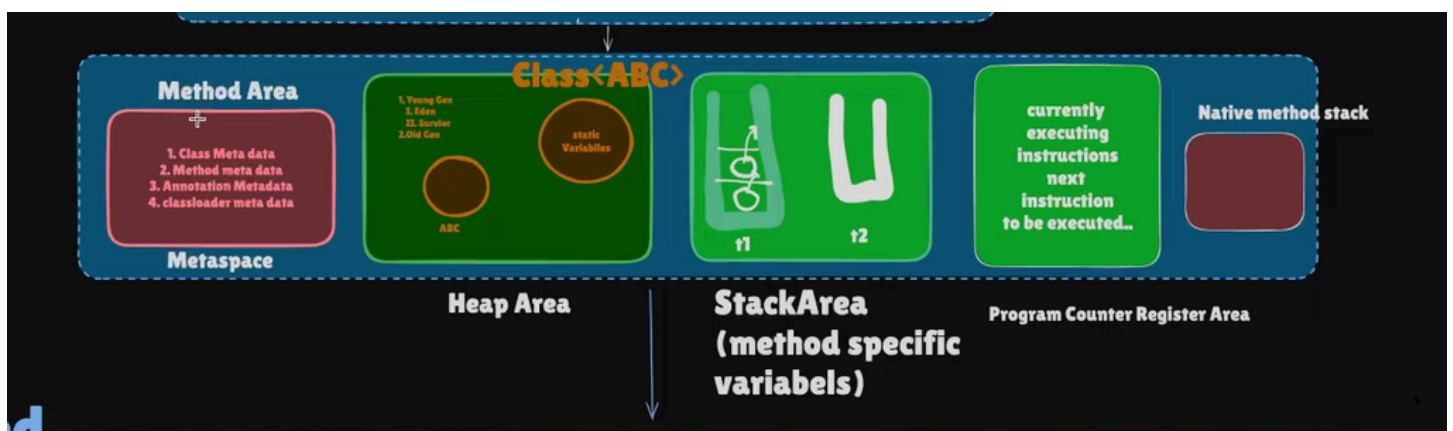


2. **Runtime Data Areas (Memory Areas)**

JVM divides memory into several regions:

- **Method Area / Class Area** (non-heap):
 - Stores class metadata (class name, fields, methods, constants, annotations, etc.).
 - Also stores bytecode of methods (including static and instance methods, constructors).
 - **Java 8 change**: Replaced **Permanent Generation (PermGen)** (fixed-size, part of heap, caused `OutOfMemoryError` easily) with **Metaspace** (native memory, dynamically resizable, grows automatically, better performance and fewer OOM errors).
 - Metaspace holds class metadata, method bytecode, annotations, reflection info, etc.
- **Heap Area** (main runtime memory):
 - Stores all objects and their instance variables.
 - **Static variables** are stored here (in the Class object) from Java 8 onward (earlier in PermGen/Method Area).
 - When a class is loaded, JVM creates a **Class** object (of type `java.lang.Class`) in the heap to represent that class → this holds static variables and class info.
 - Heap is divided for garbage collection efficiency:
 - **Young Generation** (Eden + Survivor spaces) → New/short-lived objects.
 - **Old Generation** (Tenured) → Long-lived/promoted objects.
- **Java Stacks** (one per thread):
 - Stores frames for each method call (local variables, parameters, intermediate results, operand stack).
 - Each thread has its own private stack.
 - Method calls push/pop frames (LIFO structure).

- **PC Register** (Program Counter Register) – one per thread:
 - Tracks the address of the currently executing instruction and the next one.
 - Very important for thread execution control.
- **Native Method Stack**:
 - For native (non-Java) methods (e.g., written in C/C++).



3. **Execution Engine**

- Executes the bytecode.
- Key components:
 - **Interpreter** → Executes bytecode line-by-line (converts to machine code on the fly) → Slow for repeated code.
 - **JIT Compiler** (Just-In-Time) → Identifies "hotspots" (frequently executed code/loops) → Compiles them once to native machine code → Caches it for faster repeated execution → Greatly improves performance.
 - **Garbage Collector (GC)** → Automatically reclaims memory from unreachable objects (mainly in heap) → Manages memory efficiently (e.g., Young/Old Gen collectors).
 - **Java Native Interface (JNI)** → Allows Java to call native code (C/C++) and use native libraries (e.g., OS-specific features, .dll/.so files).

- Other helpers: Thread manager, etc.

Key Java Version Changes Mentioned

- **Java 8**: Replaced PermGen with Metaspace → Removed fixed-size issues, moved class metadata to native memory.
- **Java 9+**: Modular system (java.base, etc.) replaced rt.jar → Platform ClassLoader handles modules instead of extensions folder.

This architecture makes Java secure, portable, performant (thanks to JIT + GC), and memory-managed automatically.

The video emphasizes this topic is very important for interviews and explains every step in detail with examples.



Heap Area (main runtime memory):

* Stores all objects and their instance variables., so like what is meta data of the class , like my confusion is if we have a class , what will be stored in meta data storage area , , what will be stored in Class class object ,what will be stored in heap ? of a class

Let's clear up your confusion step by step with a concrete example.

Suppose you have this simple class:

```
```java
```

```
public class Person {

 private String name; // instance variable

 private int age; // instance variable
```

```

 public static int count = 0; // static variable
```

```

 public Person(String name, int age) {

 this.name = name;

 this.age = age;

 count++;

 }
```

```

 public void sayHello() {

 System.out.println("Hi, I'm " + name);

 }
```

```
}
```

```
```
```

When the JVM loads this class (during class loading / linking / initialization), here's **exactly** what gets stored **where** (in modern JVMs, Java 8+):

1. Metaspace (Method Area / Class Metadata Area) — Native (off-heap) memory

This is **not** part of the Java heap. It stores the **blueprint / structural description** of the class (class metadata). This data is **not** a Java object — it's internal JVM data structures.

What is stored here for the `Person` class?

- Class name (`Person`)
- Superclass (`java.lang.Object`)
- Implemented interfaces (if any)
- Field metadata:
 - `name` → type `String`, access flags (private), etc.
 - `age` → type `int`, access flags (private), etc.
- Method metadata:
 - Constructor ``<init>(String,int)`` → bytecode, signature, access flags, exceptions, etc.
 - `sayHello()` → bytecode, signature, access flags, etc.
- Constant pool (literals, method/field references, strings like "Hi, I'm ", etc.)
- Annotations (if any)
- Method bytecode for **all** methods (including constructors and static methods)
- Other internal JVM info (e.g., vtable / itable for method dispatch, reflection data structures, etc.)

→ **Important**: This is **per-class** data. It's shared by **all** instances of `Person`. It lives as long as the class is loaded (usually until the classloader is GC'ed).

2. Heap — Where actual Java objects live

When the class is loaded and **initialized**, the JVM creates **one special object** in the heap:

- An instance of **`java.lang.Class`** that **represents** the `Person` class.

```
```java
```

```
Class<Person> clazz = Person.class; // This is the reference to that heap object
```

```
```
```

What is stored inside this `java.lang.Class` object (in the heap)?

- This is a normal Java object of type `java.lang.Class<Person>`.
- It contains **references** and runtime data that Java code can access via reflection, such as:
 - `getName()` → returns "Person"
 - `getFields()`, `getMethods()`, `getConstructors()` → returns `Field[]`, `Method[]`, etc.
 - `getDeclaredFields()` etc.
 - The static field values (like `Person.count`)
 - Other reflection-related caches

→ **Crucially**: The **actual values** of **static variables** (like `count = 0`) are stored **inside this `Class` object** in the heap (not in Metaspace).

So:

- `Person.count` lives inside the heap object `Person.class`
- But the **description** of the field `count` (name, type=int, static, etc.) lives in Metaspace.

3. Summary Table: One Class → Where Things Live

| What | Location | Why / Notes |
|--|--|---|
| ----- ----- ----- | | |
| ----- | | |
| Class structure & description (name, fields list, methods list, signatures, constant pool, bytecode of methods, annotations, vtable, etc.) | Metaspace | Internal JVM metadata (native memory, not Java objects). Shared for all instances. |
| Bytecode of all methods | Metaspace | Stored once per class (including static & instance methods). |
| The <code>java.lang.Class</code> object representing this class (<code>Person.class</code>) | Heap | Normal Java object. Used for reflection. |
| Values of static variables (<code>count</code> , static references, etc.) | Heap (inside the <code>Class</code> object) | Static fields belong to the class itself → stored in its <code>Class</code> instance. |
| Instances of the class (<code>new Person(...)</code>) | Heap | Each object gets its own copy of instance variables (<code>name</code> , <code>age</code>). |
| Instance variables (<code>name</code> , <code>age</code>) | Heap (inside each <code>Person</code> object) | Per-instance data. |

Quick Mental Model

- **Metaspace** = The **blueprint / recipe** of the class (structure, code). Think: class file parsed into efficient internal format.
- **Heap** → `java.lang.Class` object = The **living representative** of the class in the Java world (holds static state + reflection API).
- **Heap** → Your objects = The actual things you create with `new`.

So when people say "Metaspace stores class metadata", they mean the **structural / descriptive** metadata (not the runtime static values).

This separation (metadata in native Metaspace, static values in heap `Class` object) has been the case since Java 8.

Hope this resolves the confusion — let me know if you want a diagram explanation or more examples! Let's clear up your confusion step by step with a concrete example.

Suppose you have this simple class:

```
```java
public class Person {

 private String name; // instance variable

 private int age; // instance variable

 public static int count = 0; // static variable

 public Person(String name, int age) {

 this.name = name;

 this.age = age;

 count++;

 }

 public void sayHello() {

 System.out.println("Hi, I'm " + name);

 }

}
```
```

When the JVM loads this class (during class loading / linking / initialization), here's **exactly** what gets stored **where** (in modern JVMs, Java 8+):

1. Metaspace (Method Area / Class Metadata Area) — Native (off-heap) memory

This is **not** part of the Java heap. It stores the **blueprint / structural description** of the class (class metadata). This data is **not** a Java object — it's internal JVM data structures.

What is stored here for the `Person` class?

- Class name (`Person`)
- Superclass (`java.lang.Object`)
- Implemented interfaces (if any)
- Field metadata:
 - `name` → type `String`, access flags (private), etc.
 - `age` → type `int`, access flags (private), etc.
- Method metadata:
 - Constructor `<init>(String,int)` → bytecode, signature, access flags, exceptions, etc.
 - `sayHello()` → bytecode, signature, access flags, etc.
- Constant pool (literals, method/field references, strings like "Hi, I'm ", etc.)
- Annotations (if any)
- Method bytecode for **all** methods (including constructors and static methods)
- Other internal JVM info (e.g., vtable / itable for method dispatch, reflection data structures, etc.)

→ **Important**: This is **per-class** data. It's shared by **all** instances of `Person`. It lives as long as the class is loaded (usually until the classloader is GC'ed).

2. Heap — Where actual Java objects live

When the class is loaded and **initialized**, the JVM creates **one special object** in the heap:

- An instance of `java.lang.Class` that represents the `Person` class.

```
```java
```

```
Class<Person> clazz = Person.class; // This is the reference to that heap object
```

```
```
```

What is stored inside this `java.lang.Class` object (in the heap)?

- This is a normal Java object of type `java.lang.Class<Person>`.
- It contains **references** and runtime data that Java code can access via reflection, such as:
 - `getName()` → returns "Person"
 - `getFields()`, `getMethods()`, `getConstructors()` → returns `Field[]`, `Method[]`, etc.
 - `getDeclaredFields()` etc.
 - The static field values (like `Person.count`)
 - Other reflection-related caches

→ **Crucially**: The **actual values** of **static variables** (like `count = 0`) are stored **inside this `Class` object** in the heap (not in Metaspace).

So:

- `Person.count` lives inside the heap object `Person.class`
- But the **description** of the field `count` (name, type=int, static, etc.) lives in Metaspace.

3. Summary Table: One Class → Where Things Live

| What | Location | Why / Notes | |
|------|----------|-------------|--|
|------|----------|-------------|--|

|-----|-----|-----|-----|
-----|

| Class structure & description (name, fields list, methods list, signatures, constant pool, bytecode of methods, annotations, vtable, etc.) | **Metaspace** | Internal JVM metadata (native memory, not Java objects). Shared for all instances. |

| Bytecode of all methods | **Metaspace** | Stored once per class (including static & instance methods). |

| The `java.lang.Class` object representing this class (`Person.class`) | **Heap** | Normal Java object. Used for reflection. |

| Values of **static variables** (`count`, static references, etc.) | **Heap** (inside the `Class` object) | Static fields belong to the class itself → stored in its `Class` instance. |

| Instances of the class (`new Person(...)`) | **Heap** | Each object gets its own copy of instance variables (`name`, `age`). |

| Instance variables (`name`, `age`) | **Heap** (inside each `Person` object) | Per-instance data. |

Quick Mental Model

- **Metaspace** = The **blueprint / recipe** of the class (structure, code). Think: class file parsed into efficient internal format.
- **Heap** → `java.lang.Class` object = The **living representative** of the class in the Java world (holds static state + reflection API).
- **Heap** → Your objects = The actual things you create with `new`.

So when people say "Metaspace stores class metadata", they mean the **structural / descriptive** metadata (not the runtime static values).

This separation (metadata in native Metaspace, static values in heap `Class` object) has been the case since Java 8.

Hope this resolves the confusion — let me know if you want a diagram explanation or more examples!

what i understood is that metadata is stored in metaspace but to access it we have to use the Class class object , like the getConstructor and getFields are metadata of the class ,but for the user usage like for our program to use it is stored in Class class object but for the execution it is stored in meta data space

what i understood is that metadata is stored in metaspace but to access it we have to use the Class class object , like the getConstructor and getFields are metadata of the class ,but for the user usage like for our program to use it is stored in Class class object but for the execution it is stored in meta data space

Yes, your understanding is **mostly correct**, and you're very close to how it actually works in practice (especially in modern HotSpot JVM, Java 8+). Let me clarify it precisely and remove any remaining confusion.

Two Different Kinds of "Metadata" / Class Information

1. **Internal JVM Class Metadata** (what lives in **Metaspace**)

- This is **native (off-heap) data structures** managed entirely by the JVM.
- It includes:
 - Class name, superclass, interfaces
 - Field descriptors (names, types, access flags for both instance & static fields)
 - Method descriptors (names, signatures, bytecode, access flags)
 - Constant pool
 - Annotations
 - vtables/itables (for method dispatch)
 - Other runtime structures needed for verification, linking, reflection support, etc.
- **This data is not a Java object** — it's internal JVM bookkeeping.
- **Purpose**: The JVM uses this directly during execution (creating objects, calling methods, verifying bytecode, etc.). Your program **cannot** access this raw form.

2. **The `java.lang.Class` object** (lives in the **Heap**)

- When a class is loaded and initialized, the JVM creates **exactly one** instance of `java.lang.Class` in the **heap** to represent that class.
- Example: `Person.class` or `Class.forName("com.example.Person")` returns this heap object.
- This is a **real Java object** (of type `java.lang.Class<Person>`).
- It acts as the **public, reflection-friendly gateway** to the class.
- What it contains / exposes:
 - References to `Field[]`, `Method[]`, `Constructor[]` objects (these are also heap objects)
 - The **actual values** of static fields (e.g., `Person.count` is stored inside this object or in structures directly associated with it)
 - Methods like `getFields()`, `getConstructors()`, `getMethods()`, `newInstance()`, etc.
 - **Purpose**: This is what **your Java code** uses when doing reflection or accessing statics. It's the bridge between your program and the internal metadata.

How They Connect (Your Key Insight)

- The `java.lang.Class` object in the heap **references / points to** much of the internal metadata stored in Metaspace.
- When you call `Person.class.getConstructors()`:
 - The JVM looks at the heap `Class` object.
 - It uses pointers/references from that object to the **Metaspace metadata**.
 - It constructs `java.lang.reflect.Constructor` objects (heap objects) on the fly based on the Metaspace data.
 - Returns them to your code.
- Similarly:
 - `getFields()` → creates `Field` objects from Metaspace field descriptors.
 - Static field access (`Person.count`) → reads/writes the value stored in/associated with the heap `Class` object (not in Metaspace).

Quick Summary Table

| • Aspect | Stored In | Accessible From Java Code? | Used For Execution? |
|---|-----------------------------------|------------------------------------|--------------------------|
| Example Content | | | |
| • ----- ----- ----- ----- ----- | | | |
| • Raw class structure & bytecode | **Metaspace** | No (internal only) | Yes (by JVM) |
| Bytecode, constant pool, field/method descriptors | | | |
| • Reflection API objects | **Heap** | Yes | No (reflection overhead) |
| `Method`, `Field`, `Constructor` instances | | | |
| • Static field values | **Heap** (tied to `Class` object) | Yes | Yes |
| `Person.count = 42` value | | | |
| • <code>java.lang.Class`</code> instance itself | **Heap** | Yes (<code>Person.class`</code>) | Indirectly |
| The entry point for reflection & statics | | | |

Bottom Line

- **For JVM execution** (normal method calls, object creation, etc.) → JVM directly uses **Metaspace metadata** (fast, native).
- **For your program / user code** (especially reflection, introspection, or static access) → everything routes through the **heap `java.lang.Class`` object**, which acts as a facade/proxy to the Metaspace data.

So yes — metadata (the detailed blueprint) is in Metaspace for fast JVM-internal use, but to **access or use it from our code** (via reflection or statics), we go through the `Class`` object in the heap.

This design gives the best of both worlds: efficient runtime + safe, reflective access from Java. Great question — you nailed the distinction! If you want code examples showing reflection vs normal access, let me know. Yes, your understanding is **mostly correct**, and you're very close to how it actually works in practice (especially in modern HotSpot JVM, Java 8+). Let me clarify it precisely and remove any remaining confusion.

Two Different Kinds of "Metadata" / Class Information

1. **Internal JVM Class Metadata** (what lives in **Metaspace**)

- This is **native (off-heap) data structures** managed entirely by the JVM.
- It includes:
 - Class name, superclass, interfaces
 - Field descriptors (names, types, access flags for both instance & static fields)
 - Method descriptors (names, signatures, bytecode, access flags)
 - Constant pool
 - Annotations
 - vtables/itables (for method dispatch)
 - Other runtime structures needed for verification, linking, reflection support, etc.
- **This data is not a Java object** — it's internal JVM bookkeeping.
- **Purpose**: The JVM uses this directly during execution (creating objects, calling methods, verifying bytecode, etc.). Your program **cannot** access this raw form.

2. **The `java.lang.Class` object** (lives in the **Heap**)

- When a class is loaded and initialized, the JVM creates **exactly one** instance of `java.lang.Class` in the **heap** to represent that class.
 - Example: `Person.class` or `Class.forName("com.example.Person")` returns this heap object.
- This is a **real Java object** (of type `java.lang.Class<Person>`).
- It acts as the **public, reflection-friendly gateway** to the class.
- What it contains / exposes:
 - References to `Field[]`, `Method[]`, `Constructor[]` objects (these are also heap objects)
 - The **actual values** of static fields (e.g., `Person.count` is stored inside this object or in structures directly associated with it)

- Methods like ``getFields()``, ``getConstructors()``, ``getMethods()``, ``newInstance()``, etc.
- **Purpose**: This is what **your Java code** uses when doing reflection or accessing statics. It's the bridge between your program and the internal metadata.

How They Connect (Your Key Insight)

- The ``java.lang.Class`` object in the heap **references / points to** much of the internal metadata stored in Metaspace.
- When you call ``Person.class.getConstructors()``:
 - The JVM looks at the heap ``Class`` object.
 - It uses pointers/references from that object to the **Metaspace metadata**.
 - It constructs ``java.lang.reflect.Constructor`` objects (heap objects) on the fly based on the Metaspace data.
 - Returns them to your code.
- Similarly:
 - ``getFields()`` → creates ``Field`` objects from Metaspace field descriptors.
 - Static field access (``Person.count``) → reads/writes the value stored in/associated with the heap ``Class`` object (not in Metaspace).

Quick Summary Table

| Aspect | Stored In | Accessible From Java Code? | Used For Execution? |
|---|-----------|----------------------------|--------------------------|
| Example Content | | | |
| Raw class structure & bytecode | Metaspace | No (internal only) | Yes (by JVM) |
| Bytecode, constant pool, field/method descriptors | | | |
| Reflection API objects | Heap | Yes | No (reflection overhead) |
| Method, Field, Constructor instances | | | |

| | | | | | | |
|--|--------------------------------------|---|----------------------------------|-----|-----|--|
| Static field <code>Person.count</code> value | <code>Person.count = 42</code> value | <code>Person.class</code> instance itself | <code>Person.class</code> object | Yes | Yes | |
| <code>Person.class</code> instance itself | <code>Person.class</code> object | Yes | Indirectly | The | | |
| entry point for reflection & statics | | | | | | |

Bottom Line

- **For JVM execution** (normal method calls, object creation, etc.) → JVM directly uses **Metaspace metadata** (fast, native).
- **For your program / user code** (especially reflection, introspection, or static access) → everything routes through the **heap `java.lang.Class` object**, which acts as a facade/proxy to the Metaspace data.

So yes — metadata (the detailed blueprint) is in Metaspace for fast JVM-internal use, but to **access or use it from our code** (via reflection or statics), we go through the `Class` object in the heap.

This design gives the best of both worlds: efficient runtime + safe, reflective access from Java. Great question — you nailed the distinction! If you want code examples showing reflection vs normal access, let me know.