

✓ 5. @MethodSource

🔗 **Supplies arguments** using a **static method** that returns a `Stream`, `Collection`, or array of values.

✓ 2. How it Works

- The `@MethodSource("stringProvider")` tells JUnit to use the values returned by the `stringProvider()` method as parameters.
- `stringProvider()` must be **static** (unless using `@TestInstance(Lifecycle.PER_CLASS)`)
- Can return: `Stream`, `List`, `Set`, `Collection`, array, etc.

✓ 5. Benefits of @MethodSource

| Benefit | Description |
|--------------------------|--|
| Dynamic Data | You can generate or compute data before passing to the test. |
| More Flexible than CSV | Not limited to flat strings — supports objects, logic, etc. |
| No External Files Needed | All logic stays in Java code. |
| Reusability | Share sources between multiple test methods. |

Would you like an example using custom objects or more complex types with `@MethodSource`?

Sure! Here are **several extended examples** of `@MethodSource` in JUnit 5, showing different use cases including:

1. **Single argument**
 2. **Multiple arguments**
 3. **Custom objects**
 4. **Using a list**
 5. **Using method from another class**
-

✓ Example 1: Single Argument

```
@ParameterizedTest
@MethodSource("stringProvider")
```

```
void testSingleString(String input) {
    assertNotNull(input);
}

static Stream<String> stringProvider() {
    return Stream.of("JUnit", "Test", "MethodSource");
}
```

📌 Output:

- "JUnit"
 - "Test"
 - "MethodSource"
-

✅ Example 2: Multiple Arguments

```
@ParameterizedTest
@MethodSource("nameAgeProvider")
void testWithMultipleArgs(String name, int age) {
    assertTrue(age >= 0);
}

static Stream<Arguments> nameAgeProvider() {
    return Stream.of(
        Arguments.of("Alice", 25),
        Arguments.of("Bob", 30),
        Arguments.of("Charlie", 22)
    );
}
```

📌 Output:

| name | age |
|---------|-----|
| Alice | 25 |
| Bob | 30 |
| Charlie | 22 |

✅ Example 3: Custom Object Parameters

Suppose you have a `Person` class:

```

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

@ParameterizedTest
@MethodSource("personProvider")
void testPerson(Person person) {
    assertNotNull(person.name);
    assertTrue(person.age > 0);
}

static Stream<Person> personProvider() {
    return Stream.of(
        new Person("Alice", 30),
        new Person("Bob", 22)
    );
}

```

📌 Output:

- Person(name=Alice, age=30)
- Person(name=Bob, age=22)

✅ Example 4: Using a **List** Instead of a Stream

```

@ParameterizedTest
@MethodSource("provideColors")
void testColors(String color) {
    assertTrue(color.length() > 0);
}

static List<String> provideColors() {
    return List.of("Red", "Green", "Blue");
}

```

📌 Output:

- Red
- Green

- Blue

✅ Example 5: Method in Another Class

You can move your provider to a **different class**:

```
// In Helper.java
public class Helper {
    public static Stream<Arguments> userProvider() {
        return Stream.of(
            Arguments.of("admin", true),
            Arguments.of("guest", false)
        );
    }
}

// In your test class
@ParameterizedTest
@MethodSource("Helper#userProvider")
void testUserRoles(String username, boolean isAdmin) {
    assertNotNull(username);
}
```

Would you like to see how to use `@MethodSource` with JUnit assertions like `assertEquals`, or combine it with other annotations like `@DisplayName`?

Great request! Here are full examples of using **arrays**, **HashMap**, **Set**, and **Queue** as `@MethodSource` sources in JUnit 5.

✅ 1. Using an Array as a Method Source

```
@ParameterizedTest
@MethodSource("arrayProvider")
void testArrayValues(String input) {
    assertNotNull(input);
}

static String[] arrayProvider() {
    return new String[]{"Java", "JUnit", "Test"};
}
```

🔗 Output:

- Java
 - JUnit
 - Test
-

✓ 2. Using a HashMap as a Method Source

You can't return a `Map` directly, but you can return a `Stream<Arguments>` from its entries.

```
@ParameterizedTest
@MethodSource("mapProvider")
void testMapEntries(String key, Integer value) {
    assertTrue(value > 0);
}

static Stream<Arguments> mapProvider() {
    Map<String, Integer> data = Map.of(
        "Alice", 30,
        "Bob", 25,
        "Charlie", 35
    );
    return data.entrySet().stream()
        .map(entry -> Arguments.of(entry.getKey(), entry.getValue()));
}
```

🔴 Output:

| key | value |
|---------|-------|
| Alice | 30 |
| Bob | 25 |
| Charlie | 35 |

✓ 3. Using a Set as a Method Source

```
@ParameterizedTest
@MethodSource("setProvider")
void testSetValues(String item) {
    assertFalse(item.isEmpty());
}

static Set<String> setProvider() {
```

```
    return Set.of("One", "Two", "Three");  
}
```

📌 Output (order not guaranteed because `Set` is unordered):

- One
 - Two
 - Three
-

✓ 4. Using a Queue as a Method Source

```
@ParameterizedTest  
@MethodSource("queueProvider")  
void testQueueValues(String item) {  
    assertNotNull(item);  
}  
  
static Queue<String> queueProvider() {  
    Queue<String> queue = new LinkedList<>();  
    queue.add("First");  
    queue.add("Second");  
    queue.add("Third");  
    return queue;  
}
```

📌 Output (FIFO order):

- First
 - Second
 - Third
-

These examples show how flexible `@MethodSource` is for supplying test data from various Java collection types. Would you like a combined example that uses multiple types in a single provider?

40

You said:

give more examples on hash map

ChatGPT said:

Absolutely! Here are more detailed and varied examples using `HashMap` as a data source with `@MethodSource` in JUnit 5. Since `HashMap` itself isn't directly iterable by `@MethodSource`, we convert it to `Stream<Arguments>` using its `entrySet()`.

✓ Example 1: Basic HashMap Test

```
@ParameterizedTest
@MethodSource("basicMapProvider")
void testBasicMap(String key, Integer value) {
    assertNotNull(key);
    assertTrue(value > 0);
}

static Stream<Arguments> basicMapProvider() {
    Map<String, Integer> map = new HashMap<>();
    map.put("A", 1);
    map.put("B", 2);
    map.put("C", 3);
    return map.entrySet().stream()
        .map(e -> Arguments.of(e.getKey(), e.getValue()));
}
```

🔗 Output:

| key | value |
|-----|-------|
| A | 1 |
| B | 2 |
| C | 3 |

✓ Example 2: Map with Complex Values (List)

```
@ParameterizedTest
@MethodSource("mapWithListProvider")
void testMapWithList(String category, List<String> items) {
    assertNotNull(category);
    assertFalse(items.isEmpty());
}

static Stream<Arguments> mapWithListProvider() {
    Map<String, List<String>> map = new HashMap<>();
    map.put("fruits", List.of("apple", "banana"));
    map.put("vegetables", List.of("carrot", "peas"));
}
```

```

        map.put("grains", List.of("rice", "wheat"));

        return map.entrySet().stream()
            .map(e -> Arguments.of(e.getKey(), e.getValue()));
    }

```

✚ Output:

| category | items |
|------------|-----------------|
| fruits | [apple, banana] |
| vegetables | [carrot, peas] |
| grains | [rice, wheat] |

✓ Example 3: Map with Boolean Flags

```

@ParameterizedTest
@MethodSource("booleanMapProvider")
void testFeatureFlags(String feature, boolean isEnabled) {
    assertNotNull(feature);
}

static Stream<Arguments> booleanMapProvider() {
    Map<String, Boolean> featureFlags = Map.of(
        "Login", true,
        "Signup", false,
        "DarkMode", true
    );

    return featureFlags.entrySet().stream()
        .map(e -> Arguments.of(e.getKey(), e.getValue()));
}

```

✚ Output:

| feature | isEnabled |
|----------|-----------|
| Login | true |
| Signup | false |
| DarkMode | true |

✓ Example 4: Map with Custom Object Values

Assume you have a custom class `UserInfo`:

```
class UserInfo {
    String email;
    int age;

    UserInfo(String email, int age) {
        this.email = email;
        this.age = age;
    }
}
```

Then:

```
@ParameterizedTest
@MethodSource("customObjectMapProvider")
void testUserInfo(String username, UserInfo info) {
    assertTrue(info.age > 18);
    assertTrue(info.email.contains("@"));
}

static Stream<Arguments> customObjectMapProvider() {
    Map<String, UserInfo> users = Map.of(
        "alice", new UserInfo("alice@example.com", 25),
        "bob", new UserInfo("bob@example.com", 30)
    );

    return users.entrySet().stream()
        .map(e -> Arguments.of(e.getKey(), e.getValue()));
}
```

✂ Output:

| username | email | age |
|----------|-------------------|-----|
| alice | alice@example.com | 25 |
| bob | bob@example.com | 30 |

Would you like an example of using nested maps (e.g., `Map<String, Map<String, Integer>>`) or combining multiple maps into a single test source?

Great observation — you're thinking in a very functional and modern Java way. And yes, you **can use** `collect(Collectors.toMap(...))`, but **not as a replacement for returning `Stream<Arguments>`**, which `@MethodSource` requires.

Let me clarify both **what you can** and **can't** do in this context:

✓ What You Can Do with `Collectors.toMap()`

You can **build** the `Map` using a stream and `Collectors.toMap()` if your data starts from another collection or stream. Example:

```
static Stream<Arguments> customObjectMapProvider() {
    // Pretend we're building users from a List or similar
    List<String> names = List.of("alice", "bob");

    Map<String, UserInfo> users = names.stream()
        .collect(Collectors.toMap(
            name -> name,
            name -> new UserInfo(name + "@example.com", name.equals("alice") ? 25 : 30)
        ));

    return users.entrySet().stream()
        .map(e -> Arguments.of(e.getKey(), e.getValue()));
}
```

✓ This works well and is more dynamic if you're building the map from a data source.

✗ What You Cannot Do

You **cannot return a `Map`** from the provider method directly:

```
// ✗ INVALID: @ParameterizedTest expects Stream<Arguments>, not a Map
static Map<String, UserInfo> customObjectMapProvider() {
    return Map.of("alice", new UserInfo(...), ...);
}
```

JUnit won't know how to map the key/value to parameters without a `.map()` converting them to `Arguments`.

✓ Bottom Line

Yes, you **can use `Collectors.toMap(...)` inside your `@MethodSource`** to build the map more flexibly, but you **must still return a `Stream<Arguments>`** by mapping the map entries.

Would you like a version that builds users from a CSV file or JSON using streams and maps?