

2) Difference between *method declaration* and *method body*

- Method declaration (signature): the header that tells the compiler the method's name, return type, parameter list, and modifiers. Example parts: `public static int sum(int a, int b)` — this is the declaration/signature.
- Method body: the block of code inside `{ ... }` that implements the behavior (the statements that run when the method is called). Example: `{ return a + b; }`.

3) What keyword is used to change the access level of a method?

Access modifiers (keywords) are used: `public`, `private`, `protected`. (If none is given the method is package-private / default access — there is no keyword for that.)

4) Another word for describing the access level of a method

Common synonyms: visibility or access specifier (sometimes called access modifier).

5) Explain the scope of each variable in the code

```
public class ScopeExample {
    public static void main(String[] args) {
        int var1;
        for (int var2 = 0; var2 < 5; var2++) {
            method1();
        }
    }
    public static void method1() {
        int var3;
        for (int var4 = 0; var4 < 2; var4++) {
            var3 += 1;
        }
    }
}
```

- `var1` — local to `main`. Its scope is the entire `main` method (from declaration to end of `main`). It cannot be used by other methods.
- `var2` — declared in the `for` loop header inside `main`; its scope is limited to the loop body and loop header only. It does not exist outside the loop.
- `var3` — local to `method1`. Its scope is the whole `method1` method (from declaration to end of method). It cannot be used by `main`.
- `var4` — declared in the `for` loop header inside `method1`; its scope is limited to that loop (loop header/body).

Important note / bug in the code: `var3` is used (`var3 += 1;`) before being initialized — local variables must be initialized before use. That code will not compile. Also `var1` is declared but never initialized or used — that's allowed but likely a logic issue.

6) Method declarations (as requested)

Each is a class method (static) and callable by any method (public).

a) `getVowels` — requires a `String` parameter, returns an `int`:

```
public static int getVowels(String s) {
    // implementation here
    return 0; // placeholder
}
```

b) `extractDigit` — requires an `int` parameter, returns an `int`:

```
public static int extractDigit(int n) {
    // implementation here
    return 0; // placeholder
}
```

c) `insertString` — requires a `String` parameter and an `int` parameter, returns a `String`:

```
public static String insertString(String s, int pos) {
```

```
// implementation here  
return s; // placeholder  
}
```

(If you prefer just the signature without bodies, remove the braces and put a semicolon only for abstract/interface methods — but in a normal class provide a body.)

7a) How does the compiler distinguish one method from another?

By the method signature: method name plus parameter list (number, types, and order of parameters). This is called the method's signature. The return type is not part of the signature for overloading resolution.

7b) Can two methods in the same class have the same name? Explain.

Yes — if they have different parameter lists (different number or types/order of parameters). This is method overloading. Two methods with identical names *and* identical parameter lists are not allowed (would be a duplicate).

8a) What is the return statement used for?

To exit a method and (if the method declares a return type other than `void`) send a value back to the caller.

8b) How many values can a return statement send back to the calling statement?

One value. (That one value can be a reference to an object which itself may encapsulate many values, but a single expression is returned.)

8c) How is the declaration of a method returning a value different from a method that does not return a value?

- A method that returns a value states the return type in its declaration (e.g., `int`, `String`, `double`).
- A method that does not return a value is declared with `void`.

Example:

```
public int foo() { ... }    // returns an int
public void bar() { ... }  // returns nothing
```

9) Find and explain the error in the code below

```
public class MethodCallExample {
    public static void main(String[] args) {
        int num;
        doSomething();
        num = doSomething();
    }
}
```

Problem: This code tries to assign the result of `doSomething()` to `num`. If `doSomething()` is declared as `void doSomething()` (a void method), it does not return a value — you cannot assign a void result to an `int`. The compiler error is typically: *"void type not allowed here"*.

Fixes:

If `doSomething` should return an `int`, change its declaration to return `int` and make sure it returns an `int` value:

```
public static int doSomething() { return 42; }
```

-

Or, if `doSomething` is correctly void, remove the assignment:

```
doSomething(); // ok
// no num = doSomething();
```

-

Also ensure `num` is initialized before use if you later use it.

True / False (with concise explanations)

For each item: state true/false and why.

a) Breaking a task down into methods is called procedural abstraction.

True. Procedural abstraction (or decomposition) is creating methods to hide details and present a simpler interface.

b) A method call consists of the method declaration in an assignment statement.

False. A *method call* (invocation) is code that invokes the method (e.g., `x = f(3);` or `f(3);`). The *method declaration* is the method's definition — not part of the call.

c) A void method must return a value.

False. A void method does not return a value. It may use `return;` to exit early but cannot return a value.

d) An access modifier declares the return type of a method.

False. Access modifiers (`public`, `private`, etc.) control visibility. The return type is a separate part of the declaration (e.g., `int`, `String`, `void`).

e) The keyword `static` declares a method is a class method.

True. `static` makes a method belong to the class rather than an instance (class method).

f) Method parameters are enclosed by braces `{ }`.

False. Parameters are enclosed by parentheses `(...)`. Braces `{ }` enclose the method body.

g) Local variables can be used by any method in a class.

False. Local variables are confined to the method or block where they are declared.

h) The value of an argument passed to a method can be changed in an assignment statement in the method.

False (in general). Assigning to the parameter variable inside the method changes only the local copy of the reference/primitive — it does not change the caller's variable for primitives. For object references, you can modify the object's internal state (e.g., `obj.setX(3)`), but you cannot make the caller's reference point to a different object by reassigning the parameter. So the statement as given is misleading -> false, with the above nuance.

i) Method overloading means that an application contains more than 10 methods.

False. Overloading means multiple methods with the same name but different parameter lists.

j) The `return` statement is used to send a value back to the calling statement.

True. That is its primary purpose (for non-void methods). It also exits the method.

k) The precondition of a method states the data types of the method's parameters.

False. Preconditions describe what must be true before calling the method (valid ranges, non-null, etc.). Parameter types are part of the method signature, not the precondition.

l) The postcondition of a method describes the way the method accomplishes its task.
False. Postconditions describe the state/results after the method finishes (what must be true after). The way it accomplishes it is the implementation (not the postcondition).