Java Day3

What are Java Methods?

In Java, a method is a set of statements that perform a certain action and are declared within a class.

Here's the fundamental syntax for a Java method:

```
acessSpecifier returnType methodName(parameterType1 parameterName1, parameterType2 parameterName2, ...) {
    // Method body - statements to perform a specific task
    // Return statement (if applicable)
}
```

Let's break down the components:

- `accessSpecifier`: defines the visibility or accessibility of classes, methods, and fields within a program.
- `returnType`: the data type of the value that the method returns. If the method does not return any value, the `void` keyword is used.
- `methodName`: the name of the method, following Java naming conventions.
- `parameter`: input value that the method accepts. These are optional, and a method can have zero or more parameters. Each parameter is declared with its data type and a name.
- **method body**: the set of statements enclosed in curly braces `{}` that define the task the method performs.
- **return statement**: if the method has a return type other than `void`, it must include a `return` statement followed by the value to be returned.

Here's an example of a simple Java method:

```
public class SimpleMethodExample {

    // Method that takes two integers and returns their sum
    public static int addNumbers(int a, int b) {
        int sum = a + b;
```

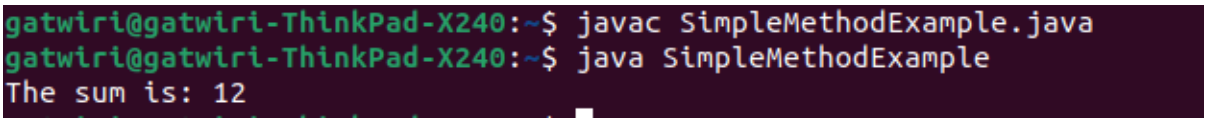
```
        return sum;
    }

    public static void main(String[] args) {
        // Calling the method and storing the result
        int result = addNumbers(5, 7);

        // Printing the result
        System.out.println("The sum is: " + result);
    }
}
```

In this example, the `addNumbers` method takes two integers as parameters (`a` and `b`), calculates their sum, and returns the result. The `main` method then calls this method and prints the result.

Compile the Java code using the terminal, using the `javac` command:



Output

Methods facilitate code reusability by encapsulating functionality in a single block. You can call that block from different parts of your program, avoiding code duplication and promoting maintainability.

Types of Access Specifiers in Java

Access specifiers control the visibility and accessibility of class members (fields, methods, and nested classes).

There are typically four main types of access specifiers: public, private, protected, and default. They dictate where and how these members can be accessed, promoting encapsulation and modularity.

Public (`public`)

This grants access to the member from **anywhere** in your program, regardless of package or class. It's suitable for widely used components like utility functions or constants.

Syntax:

```
public class MyClass {
    public int publicField;
    public void publicMethod() {
        // method implementation
    }
}
```

Example:

```
// File: MyClass.java

// A class with public access specifier
public class MyClass {

    // Public field
    public int publicField = 10;

    // Public method
    public void publicMethod() {
        System.out.println("This is a public method.");
    }

    // Main method to run the program
    public static void main(String[] args) {
        // Creating an object of MyClass
        MyClass myObject = new MyClass();

        // Accessing the public field
        System.out.println("Public Field: " + myObject.publicField);

        // Calling the public method
        myObject.publicMethod();
    }
}
```

In this example:

- The `MyClass` class is declared with the `public` modifier, making it accessible from any other class.
- The `publicField` is a public field that can be accessed from outside the class.
- The `publicMethod()` is a public method that can be called from outside the class.
- The `main` method is the entry point of the program, where an object of `MyClass` is created, and the public field and method are accessed.

```
Public Field: 10
This is a public method.
```

Private (`private`)

This confines access to the member **within the class** where it's declared. It protects sensitive data and enforces encapsulation.

Syntax:

```
public class MyClass {
    private int privateField;
    private void privateMethod() {
        // method implementation
    }
}
```

Example:

```
// File: MyClass.java

// A class with private access specifier
public class MyClass {

    // Private field
    private int privateField = 10;

    // Private method
    private void privateMethod() {
        System.out.println("This is a private method.");
```

```
    }

    // Public method to access private members
    public void accessPrivateMembers() {
        // Accessing the private field
        System.out.println("Private Field: " + privateField);

        // Calling the private method
        privateMethod();
    }

    // Main method to run the program
    public static void main(String[] args) {
        // Creating an object of MyClass
        MyClass myObject = new MyClass();

        // Accessing private members through a public method
        myObject.accessPrivateMembers();
    }
}
```

In this example:

- The `MyClass` class has a `privateField` and a `privateMethod`, both marked with the `private` modifier.
- The `accessPrivateMembers()` method is a public method that can be called from outside the class. It provides access to the private field and calls the private method.

```
Private Field: 10
This is a private method.
```

Protected (`protected`)

The `protected` access specifier is used to make members (fields and methods) accessible within the same package or by subclasses, regardless of the package. They are not accessible from unrelated classes. It facilitates inheritance while controlling access to specific members in subclasses.

Syntax:

```
public class MyClass {
    protected int protectedField;
    protected void protectedMethod() {
        // method implementation
    }
}
```

Example:

```
// File: Animal.java

// A class with protected access specifier
public class Animal {

    // Protected field
    protected String species = "Unknown"; // Initialize with a default value

    // Protected method
    protected void makeSound() {
        System.out.println("Some generic animal sound");
    }
}
```

```
// File: Dog.java

// A subclass of Animal
public class Dog extends Animal {

    // Public method to access protected members
    public void displayInfo() {
        // Accessing the protected field from the superclass
        System.out.println("Species: " + species);

        // Calling the protected method from the superclass
        makeSound();
```

```
    }
}
```

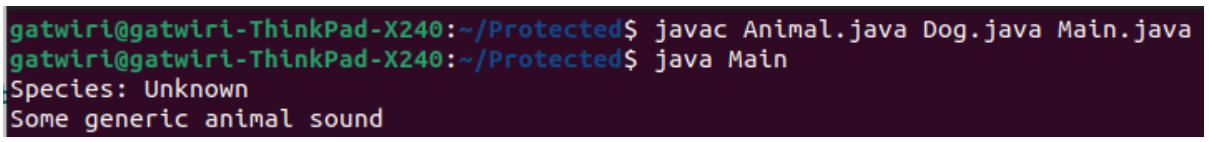
```
// File: Main.java

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        // Creating an object of Dog
        Dog myDog = new Dog();

        // Accessing protected members through a public method
        myDog.displayInfo();
    }
}
```

In this example:

- The `Animal` class has a `protected` field (`species`) and a `protected` method (`makeSound`).
- The `Dog` class is a subclass of `Animal`, and it can access the `protected` members from the superclass.
- The `displayInfo()` method in the `Dog` class accesses the protected field and calls the protected method.



Output

With the `protected` access specifier, members are accessible within the same package and by subclasses, promoting a certain level of visibility and inheritance while still maintaining encapsulation.

Default (`Package-Private`)

If no access specifier is used, the default access level is `package-private`. Members with default access are accessible within the same package, but not

outside it. It's often used for utility classes or helper methods within a specific module.

Syntax:

```
class MyClass {
    int defaultField;
    void defaultMethod() {
        // method implementation
    }
}
```

Example:

```
// File: Animal.java

// A class with default (package-private) access specifier
class Animal {
    String species = "Unknown";

    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}
```

```
// File: Main.java

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        // Creating an object of Dog
        Dog myDog = new Dog();

        // Accessing default (package-private) members through a public met
hod
        myDog.displayInfo();
    }
}
```

```
// File: Dog.java

// Another class in the same package
public class Dog {
    Animal myAnimal = new Animal();

    void displayInfo() {
        // Accessing the default (package-private) field and method
        System.out.println("Species: " + myAnimal.species);
        myAnimal.makeSound();
    }
}
```

In this example:

- The `Animal` class does not have any access modifier specified, making it default (package-private). It has a package-private field `species` and a package-private method `makeSound`.
- The `Dog` class is in the same package as `Animal`, so it can access the default (package-private) members of the `Animal` class.
- The `Main` class runs the program by creating an object of `Dog` and calling its `displayInfo` method.

When you run this program, it should output the species and the sound of the animal.

How to Choose the Right Access Specifier

- **Public:** Use for widely used components, interfaces, and base classes.
- **Private:** Use for internal implementation details and sensitive data protection.
- **Default:** Use for helper methods or components specific to a package.
- **Protected:** Use for shared functionality among subclasses, while restricting access from outside the inheritance hierarchy.

Types of Methods

In Java, methods can be categorized in two main ways:

1. Predefined vs. User-defined:

Predefined methods: These methods are already defined in the Java Class Library and can be used directly without any declaration.

Examples include `System.out.println()` for printing to the console and `Math.max()` for finding the maximum of two numbers.

User-defined methods: These are methods that you write yourself to perform specific tasks within your program. They are defined within classes and are typically used to encapsulate functionality and improve code reusability.

```
public class RectangleAreaCalculator {

    // User-defined method to calculate the area of a rectangle
    public static double calculateRectangleArea(double length, double width)
{
        double area = length * width;
        return area;
    }

    public static void main(String[] args) {
        // Example of using the method
        double length = 5.0;
        double width = 3.0;

        // Calling the method
        double result = calculateRectangleArea(length, width);

        // Displaying the result
        System.out.println("The area of the rectangle with length " + length +
" and width " + width + " is: " + result);
    }
}
```

In this example:

- `add` is a user-defined method because it's created by the user (programmer).
- The method takes two parameters (`num1` and `num2`) and returns their sum.

- The `main` method calls the `add` method with specific values, demonstrating the customized functionality provided by the user.

2. Based on functionality:

Within user-defined methods, there are several other classifications based on their characteristics:

Instance Methods:

Associated with an instance of a class. They can access instance variables and are called on an object of the class.

Here are some key characteristics of instance methods:

Access to Instance Variables:

- Instance methods have access to instance variables (also known as fields or properties) of the class.
- They can manipulate the state of the object they belong to.

Use of `this` Keyword:

- Inside an instance method, the `this` keyword refers to the current instance of the class. It's often used to differentiate between instance variables and parameters with the same name.

Non-static Context:

- Instance methods are called in the context of an object. They can't be called without creating an instance of the class.

Declaration and Invocation:

- Instance methods are declared without the `static` keyword.
- They are invoked on an instance of the class using the dot (`.`) notation.

Here's a simple example in Java to illustrate instance methods:

Example:

```
public class Dog {
    // Instance variables
    String name;
    int age;
```

```
    // Constructor to initialize the instance variables
    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Instance method to bark
    public void bark() {
        System.out.println(name + " says Woof!");
    }

    // Instance method to age the dog
    public void ageOneYear() {
        age++;
        System.out.println(name + " is now " + age + " years old.");
    }

    public static void main(String[] args) {
        // Creating instances of the Dog class
        Dog myDog = new Dog("Buddy", 3);
        Dog anotherDog = new Dog("Max", 2);

        // Calling instance methods on objects
        myDog.bark();
        myDog.ageOneYear();

        anotherDog.bark();
        anotherDog.ageOneYear();
    }
}
```

In this example:

- `bark` and `ageOneYear` are instance methods of the `Dog` class.
- They are invoked on instances of the `Dog` class (`myDog` and `anotherDog`).
- These methods can access and manipulate the instance variables (`name` and `age`) of the respective objects.

Instance methods are powerful because they allow you to encapsulate behavior related to an object's state and provide a way to interact with and modify that state.

Static Methods:

A static method belongs to the class rather than an instance of the class. This means you can call a static method without creating an instance (object) of the class. It's declared using the `static` keyword.

Static methods are commonly used for utility functions that don't depend on the state of an object. For example, methods for mathematical calculations, string manipulations, and so on.

Example:

```
public class MathOperations {
    // Static method
    public static int add(int a, int b) {
        return a + b;
    }

    // Static method
    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Abstract Methods:

These methods are declared but not implemented in a class. They are meant to be overridden by subclasses, providing a blueprint for specific functionality that must be implemented in each subclass.

Abstract methods are useful when you want to define a template in a base class or interface, leaving the specific implementation to the subclasses. Abstract methods define a contract that the subclasses must follow.

Example:

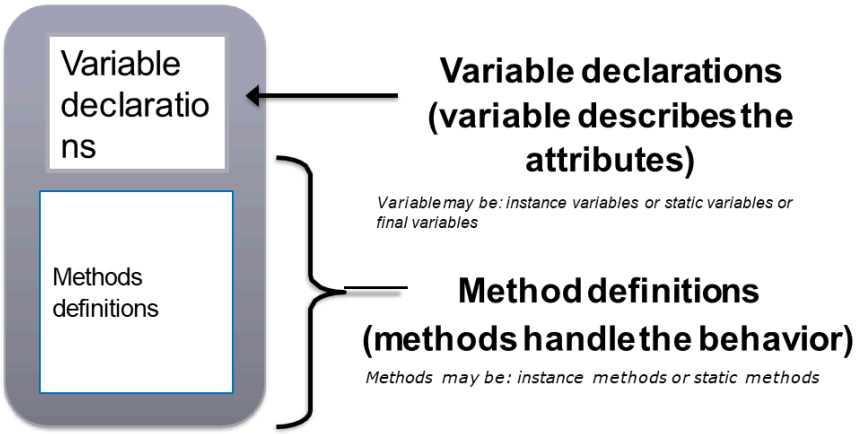
```
public abstract class Shape {
    // Abstract method
```

```
    abstract double calculateArea();
}
```

Other method types: Additionally, there are less common types like constructors used for object initialization, accessor methods (getters) for retrieving object data, and mutator methods (setters) for modifying object data.

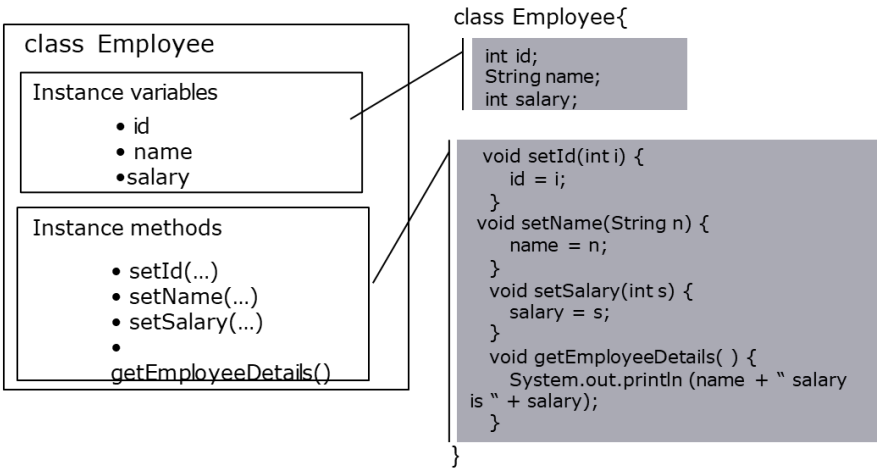
Classes

A class contains variable declarations and method definitions



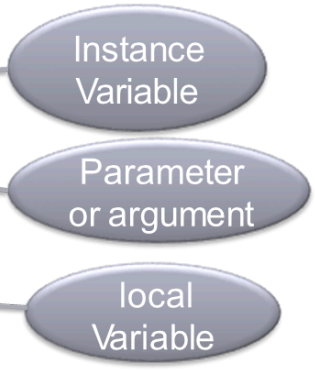
Defining a Class in java

Define an Employee class with instance variables and instance methods



Basic information about a class

```
public class Account {
   double balance;
   public void deposit( double amount
     ){ balance += amount;
   }
   public double withdraw( double amount ){
     int minimum_balance=5000;
     if (balance >= (amount+minimum_balance)){
          balance -= amount;
          return amount;
     }
     else {
          System.out.println("Insufficient Balance");
          return 0.0;
     }    }
    public double getbalance(){
                return balance;
   } }
```



Basic information about a class (Contd.).

```
    else {
        System.out.println("Insufficient Balance");
        return 0.0;
    }
  }
   public double getbalance(){
               return balance;
  }
}
```

Member variables

- The previous slide contains definition of a class called Accounts.
- A class contains members which can either be variables(fields) or methods(behaviors).
- A variable declared within a class(outside any method) is known as an **instance variable**.
- A variable declared within a method is known as **local variable**.
- Variables with method declarations are known as **parameters or arguments**.
- A class variable can also be declared as static where as a local variable cannot be static.

Objects and References

- Once a class is defined, you can declare a variable (object reference) of type class

```
Student stud1;
Employee emp1;
```

- The new operator is used to create an object of that reference type

```
Employee emp = new Employee();
```

Object reference — object

- Object references are used to store objects.
- Reference can be created for any type of classes (like concrete classes, abstract classes) and interfaces.

Objects and References (Contd.).

- The new operator,
  - Dynamically allocates memory for an object
  - Creates the object on the heap
  - Returns a reference to it
  - The reference is then stored in the variable

Employee class - Example

```
class Employee{
int id;
String name;
int salary;
void setId(int no){
id = no;
}
void setName(String n){
name = n;
}
void setSalary(int s){
salary = s;
}
void  getEmployeeDetails(){
System.out.println(name + " salary is "+ salary);
}
}
public class EmployeeDemo {
public static void main(String[] args) {
Employee emp1 = new Employee();
emp1.setId(101);
emp1.setName("John");
emp1.setSalary(12000);
emp1.getEmployeeDetails();
}
}
```

Output:

John salary is 12000

Constructors

- While designing a class, the class designer can define within the class, a special method called 'constructor'
- Constructor is automatically invoked whenever an object of the class is created
- Rules to define a constructor
  - A constructor has the same name as the class name
  - A constructor should not have a return type
  - A constructor can be defined with any access specifier (like private, public)
  - A class can contain more than one constructor, So it can be overloaded

Constructor - Example

```
class Sample{
private int id;
Sample(){
id = 101;
System.out.println("Default constructor, with ID: "+id);
}
Sample(int no){
id = no;
System.out.println("One argument constructor,with ID: "+ id);
}
}
public class ConstDemo {
public static void main(String[] args) {
Sample s1 = new Sample();
Sample s2 = new Sample(102);
}
}
```

Output:
Default constructor, with ID: 101
One argument constructor,with ID: 102

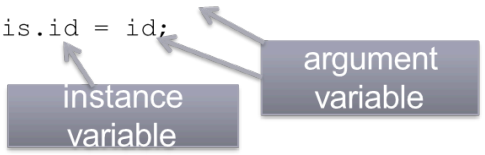
this reference keyword

- Each class member function contains an implicit reference of its class type, named this
- this reference is created automatically by the compiler
- It contains the address of the object through which the function is invoked
- Use of this keyword
  - this can be used to refer instance variables when there is a clash with local variables or method arguments
  - this can be used to call overloaded constructors from another constructor of the same class

this Reference (Contd.).

- **Ex1:**

```
void  setId (int id){
    this.id = id;
}
```



- **Ex2:**

```
class Sample{
Sample(){
this("Java"); // calls overloaded constructor
System.out.println("Default constructor ");
}
Sample(String str){
System.out.println("One argument constructor "+ str);
}
}
```

this Reference (Contd.).

- Use this.variableName to explicitly refer to the instance variable.
- Use variable Name to refer to the parameter.
- The **this** reference is implicitly used to refer to instance variables and methods.
- It **CANNOT** be used in a static method.

Static Class Members

- Static class members are the members of a class that do not belong to an instance of a class
- We can access static members directly by prefixing the members with the class name

  ClassName.staticVariable

  ClassName.staticMethod(...)

Static variables:

- Shared among all objects of the class
- Only one copy exists for the entire class to use

Static Class Members (Contd.).

- Stored within the class code, separately from instance variables that describe an individual object
- Public static final variables are global constants

Static methods:

- Static methods can only access directly the static members and manipulate a class's static variables
- Static methods cannot access non-static members(instance variables or instance methods) of the class
- Static method cant access this and super references

Static Class Members – Example

```
class StaticDemo
{
private static int a = 0;
private int b;
public void set ( int i, int j)
{
a = i; b = j;
}
public void show( )
{
System.out.println("This is static a: " + a );
System.out.println( "This is non-static b: " + b );
}
```

Static Class Members – Example (Contd.).

```
public static void main(String args[ ])
{
StaticDemo x = new StaticDemo( );
StaticDemo y = new StaticDemo( );
x.set(1, 1);
x.show( );
y.set(2, 2);
y.show( );
x.show( );
}
}
```

Output:

```
This is static a: 1
This is non-static  b: 1

This is static a: 2
This is non-static  b: 2

This is static a: 2
This is non-static  b: 1
```

Time to Think

Why is main() method static ?

Time to Think(Contd.).

- If a java application has to be executed, there has to be a starting point. main() method is supposed to be that starting point. But Java doesn't allow you to execute any method unless you create an object of the class where the method resides. Unless we execute the code, how do we create an instance?

The “static” block

- A static block is a block of code enclosed in braces, preceded by the keyword static

Ex :

```
static {
  System.out.println(“Within static block”);
}
```

- The statements within the static block are executed automatically when the class is loaded into JVM

The “static” block (Contd.).

- A class can have any number of static blocks and they can appear anywhere in the class
- They are executed in the order of their appearance in the class
- JVM combines all the static blocks in a class as single static block and executes them
- You can invoke static methods from the static block and they will be executed as and when the static block gets executed

Example on the “static” block (Contd.).

```
class StaticBlockExample {
  StaticBlockExample() {
    System.out.println("Within constructor");
  }
  static {
    System.out.println("Within 1st static block");
  }
  static void m1() {
    System.out.println("Within static m1 method");
  }
static {
    System.out.println("Within 2nd static block");
    m1();
  }
```

Example on the “static” block (Contd.).

```
  public static void main(String [] args) {
    System.out.println("Within main");
    StaticBlockExample x = new StaticBlockExample();
  }
  static {
    System.out.println("Within 3rd static block");
  }
}
```

Output:

Within 1st static block
Within 2nd static block
Within static m1 method
Within 3rd static block
Within main
Within constructor

🧠 Types of Variables in Java

1. **Instance Variables**

- Declared inside a class but **outside any method**.
- Each object of the class has its own copy.
- Typically used to store object state.

2. **Static Variables (Class Variables)**

- Declared with the `static` keyword inside a class but outside any method.
- Shared among all instances of the class.
- Used for common properties across all objects.

3. **Local Variables**

- Declared **inside a method, constructor, or block**.
- Only accessible within that method or block.
- Must be initialized before use.

🎓 Example: `Student` Class in Java

```
public class Student {
    // Instance variable
    String name;
    int age;

    // Static variable
    static String schoolName = "Greenwood High";

    // Constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method demonstrating local variable
    public void displayInfo() {
        // Local variable
        String grade = "A";
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("School: " + schoolName);
        System.out.println("Grade: " + grade);
    }
}
```