FUNCTION OVERLOADING IN JAVA

FOR CLASS 10 (SCIENCE)

METHOD

- A collection of statements that are grouped together to perform an operation
- A method has the following syntax

  modifier return_Value_Type Method_Name(list of parameters)

  { // Method body; }

- A method definition consists of a method header and a method body

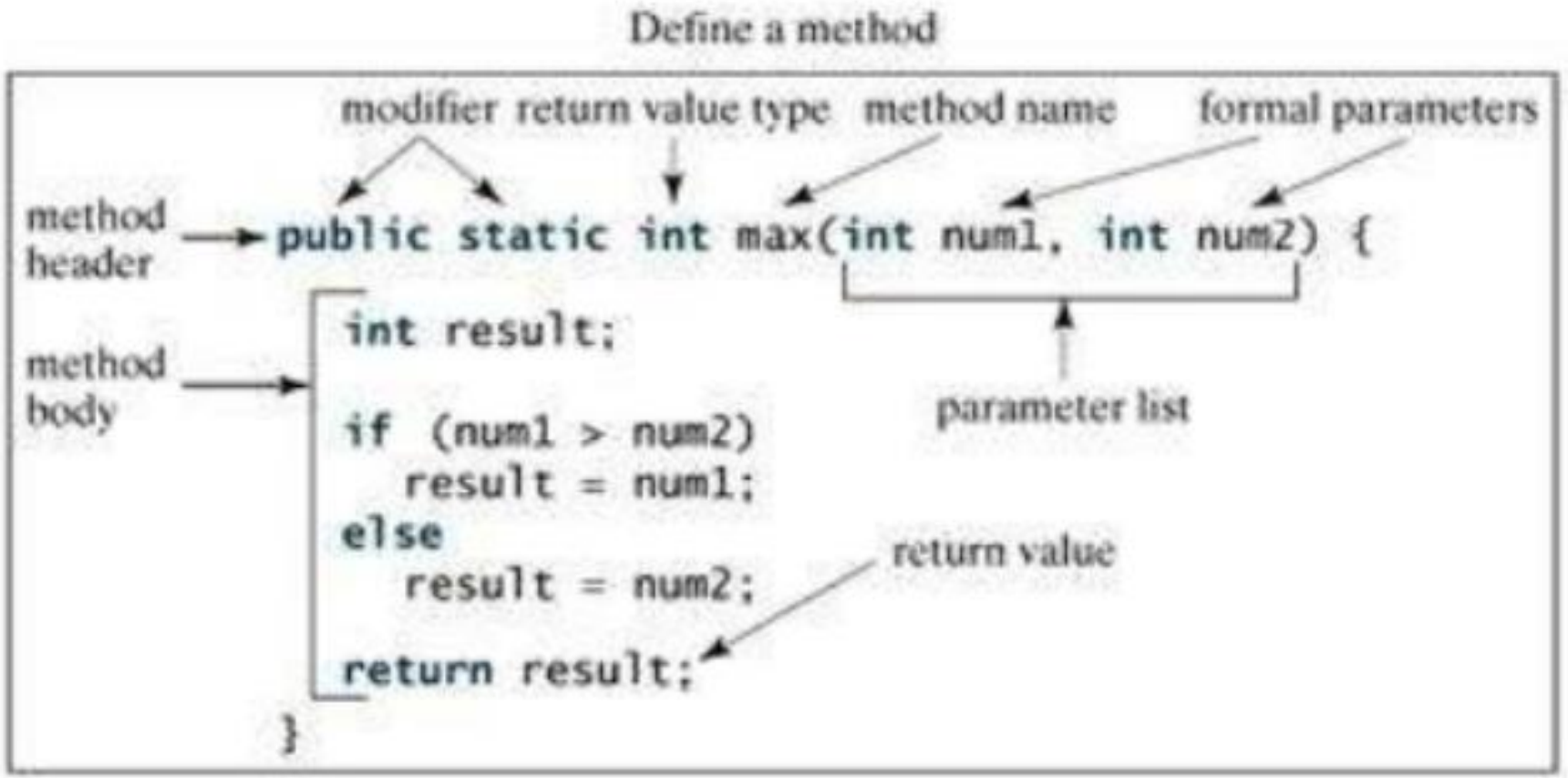
PARTS OF METHOD

1. **Modifiers:**
   - Optional
   - Tells the compiler how to call the method.
   - Defines the access type of the method.
2. **Return Types**
   - The data type of the value the method returns.
   - the return type is the keyword **void** when no value is returned

- **Method Name**
  - The actual name of the method.
  - Method name and the parameter list together constitute the method signature.

- **Parameters**
  - Act as a placeholder.
  - When a method is invoked, a value is passed to parameter.
  - This value is referred to as actual parameter or argument.
  - The parameter list refers to the type, order, and number of the parameters of a method.
  - Parameters are optional; that is,

- **Method Body**
  - The method body contains a collection of statements that define what the method does.



METHOD OVERLOADING

- A concept in Java which allows programmer to declare method with same name but different behavior
- Method with same name co-exists in same class but they must have different method signature
- Resolved using static binding in Java at compile time
- When you overload a method in Java its method signature gets changed

How to overload a method in Java

- If you have two methods with same name in one Java class with different method signature than its called overloaded method in Java
- Generally overloaded method in Java has different set of arguments to perform something based on different number of input
- Binding of overloading method occurs during compile time and overloaded calls resolved using static binding.
- To overload a Java method just changes its signature.
- To change signature, either change number of argument, type of argument or order of argument
- Since return type is not part of method signature changing return type will result in duplicate method and you will get compile time error in Java

IMPORTANT

Static and Dynamic Binding

- **Binding**-process used to link which method or variable to be called as result of their reference in code
- Two types of binding
  - Static Binding, and
  - Dynamic Binding.
- Static binding in Java occurs during Compile time while Dynamic binding occurs during Runtime.
- private, final and static methods and variables uses static binding and bonded by compiler while virtual methods are bonded during runtime based upon runtime object.
- Static binding uses class information for binding while Dynamic binding uses Object to resolve binding.
- Overloaded method are bonded using static binding while overridden methods are bonded during compile time.
  - Method override -A method in subclass has the same name & type signature as a method in its superclass.

Example

```
public class MethodOverLoading {
// Method 1
    public static int Addition(int a, int b) {
    System.out.println("Method 1 is called");
    return a + b;
    }
// Method 2
    public static int Addition(int a, int b, int c) {
    System.out.println("Method 2 is called");
    return a + b + c;
    }
    }
    public static void main(String[] args) {
    int Answer1 = Addition(5, 6); // In this case
                    Method 1 will be called
    System.out.println("Answer 1 = " + Answer1);
    System.out.println("--------------------------");
    int Answer2 = Addition(5, 6, 7); // In this case
                    Method 2 will be called
    System.out.println("Answer 2 = " + Answer2);
    }
```

Output of the code

```
Method 1 is called
Answer 1 = 11
--------------------------
Method 2 is called
Answer 2 = 18
```

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
a 10
```

In the above example – method `disp()` is overloaded based on the number of parameters – We have two methods with the name `disp` but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

```
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```

Output:

```
a
5
```

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
   public void disp(char c, int num)
   {
       System.out.println("I'm the first definition of method disp");
   }
   public void disp(int num, char c)
   {
       System.out.println("I'm the second definition of method disp" );
   }
}
class Sample3
{
   public static void main(String args[])
   {
       DisplayOverloading3 obj = new DisplayOverloading3();
       obj.disp('x', 51 );
       obj.disp(52, 'y');
   }
}
```

Output:

```
I'm the first definition of method disp
I'm the second definition of method disp
```

Types Of Access Specifiers :

In java we have four Access Specifiers and they are listed below.

1. public
2. private
3. protected
4. default(no specifier)

We look at these Access Specifiers in more detail.

Access Modifiers	Default	private	protected	public
Accessible inside the class	yes	yes	yes	yes
Accessible within the subclass inside the same package	yes	no	yes	yes
Accessible outside the package	no	no	no	yes
Accessible within the subclass outside the package	no	no	yes	yes

Private and final methods in Java

When we use *final* specifier with a method, the method cannot be overridden in any of the inheriting classes. Methods are made final due to design reasons.

Since private methods are inaccessible, they are implicitly final in Java. So adding *final* specifier to a private method doesn't add any value. It may in-fact cause unnecessary confusion.

THANK YOU