Java Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

```
public class Main {

  static void myMethod() {

    // code to be executed

  }

}
```

Example Explained

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `myMethod()` is used to print a text (the action), when it is called:

```
public class Main {

  static void myMethod() {

    System.out.println("I just got executed!");

  }


  public static void main(String[] args) {

    myMethod();

  }

}
```

A method can also be called multiple times:

```
public class Main {

  static void myMethod() {

    System.out.println("I just got executed!");

  }


  public static void main(String[] args) {

    myMethod();

    myMethod();

    myMethod();

  }
}
```

Java Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```
public class Main {

  static void myMethod(String fname) {

    System.out.println(fname + " Refsnes");

  }


  public static void main(String[] args) {

    myMethod("Liam");

    myMethod("Jenny");

    myMethod("Anja");

  }

}
```

Multiple Parameters

```
public class Main {

  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}
```

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int`, `char`, etc.) instead of `void`, and use the `return` keyword inside the method:

```
public class Main {
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

```
public class Main {
  static int myMethod(int x, int y) {

    return x + y;

  }


  public static void main(String[] args) {

    System.out.println(myMethod(5, 3));

  }

}

// Outputs 8 (5 + 3)
```

you can also store the result in a variable (recommended, as it is easier to read and maintain):

```
public class Main {

  static int myMethod(int x, int y) {

    return x + y;

  }


  public static void main(String[] args) {

    int z = myMethod(5, 3);

    System.out.println(z);

  }

}

// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use `if...else` statements inside methods:

```
public class Main {
```

```java
  // Create a checkAge() method with an integer variable called age
  static void checkAge(int age) {

    // If age is less than 18, print "access denied"
    if (age < 18) {
      System.out.println("Access denied - You are not old enough!");

    // If age is greater than, or equal to, 18, print "access granted"
    } else {
      System.out.println("Access granted - You are old enough!");
    }

  }

  public static void main(String[] args) {
    checkAge(20); // Call the checkAge method and pass along an age of 20
  }
}

// Outputs "Access granted - You are old enough!"
```

Java Method Overloading

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

```
int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)
```

Consider the following example, which has two methods that add numbers of different type:

```
public class Main {

  static int plusMethodInt(int x, int y) {

    return x + y;

  }



  static double plusMethodDouble(double x, double y) {

    return x + y;

  }



  public static void main(String[] args) {

    int myNum1 = plusMethodInt(8, 5);

    double myNum2 = plusMethodDouble(4.3, 6.26);

    System.out.println("int: " + myNum1);

    System.out.println("double: " + myNum2);

  }

}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double`:

```
static int plusMethod(int x, int y) {

  return x + y;

}



static double plusMethod(double x, double y) {

  return x + y;

}



public static void main(String[] args) {

  int myNum1 = plusMethod(8, 5);

  double myNum2 = plusMethod(4.3, 6.26);

  System.out.println("int: " + myNum1);

  System.out.println("double: " + myNum2);

}
```

Method overriding in java with example

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The `Boy` class extends `Human` class. Both the classes have a common method `void eat()`. Boy class is giving its own implementation to the `eat()` method or in other words it is overriding the `eat()` method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
   //Overridden method
   public void eat()
   {
      System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public void eat(){
      System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
      Boy obj = new Boy();
      //This will call the child class version of eat()
      obj.eat();
   }
}
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
   //Overridden method
   public void disp()
   {
        System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
        System.out.println("disp() method of Child class");
   }
   public void newMethod(){
        System.out.println("new method of child class");
   }
   public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object
         * then in this case overridden method (the method of parent class)
         *  is called.
         */
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object
         * then the overriding method (method of child class) is called.
         * This is called dynamic method dispatch and runtime polymorphism
         */
        ABC obj2 = new Demo();
        obj2.disp();
   }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).
Note: In dynamic method dispatch the object can call the overriding methods of

child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object `obj2` is calling the `disp()`. However if you try to call the `newMethod()` method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: The method xyz() is undefined for the type ABC
```

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.
   For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
   public void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   protected void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: Cannot reduce the visibility of the inherited method from
MyBaseClass
```

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
   protected void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   public void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Child class method
```

17. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
18. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
19. Binding of overridden methods happen at runtime which is known as dynamic binding.
20. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the myMethod() method of base class while `super()` calls the constructor of base class. Let's see the use of super in method Overriding.
As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

```java
class ABC{
   public void myMethod()
   {
        System.out.println("Overridden method");
   }
}
class Demo extends ABC{
   public void myMethod(){
        //This will call the myMethod() of parent class
        super.myMethod();
        System.out.println("Overriding method");
   }
   public static void main( String args[]) {
        Demo obj = new Demo();
        obj.myMethod();
   }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```