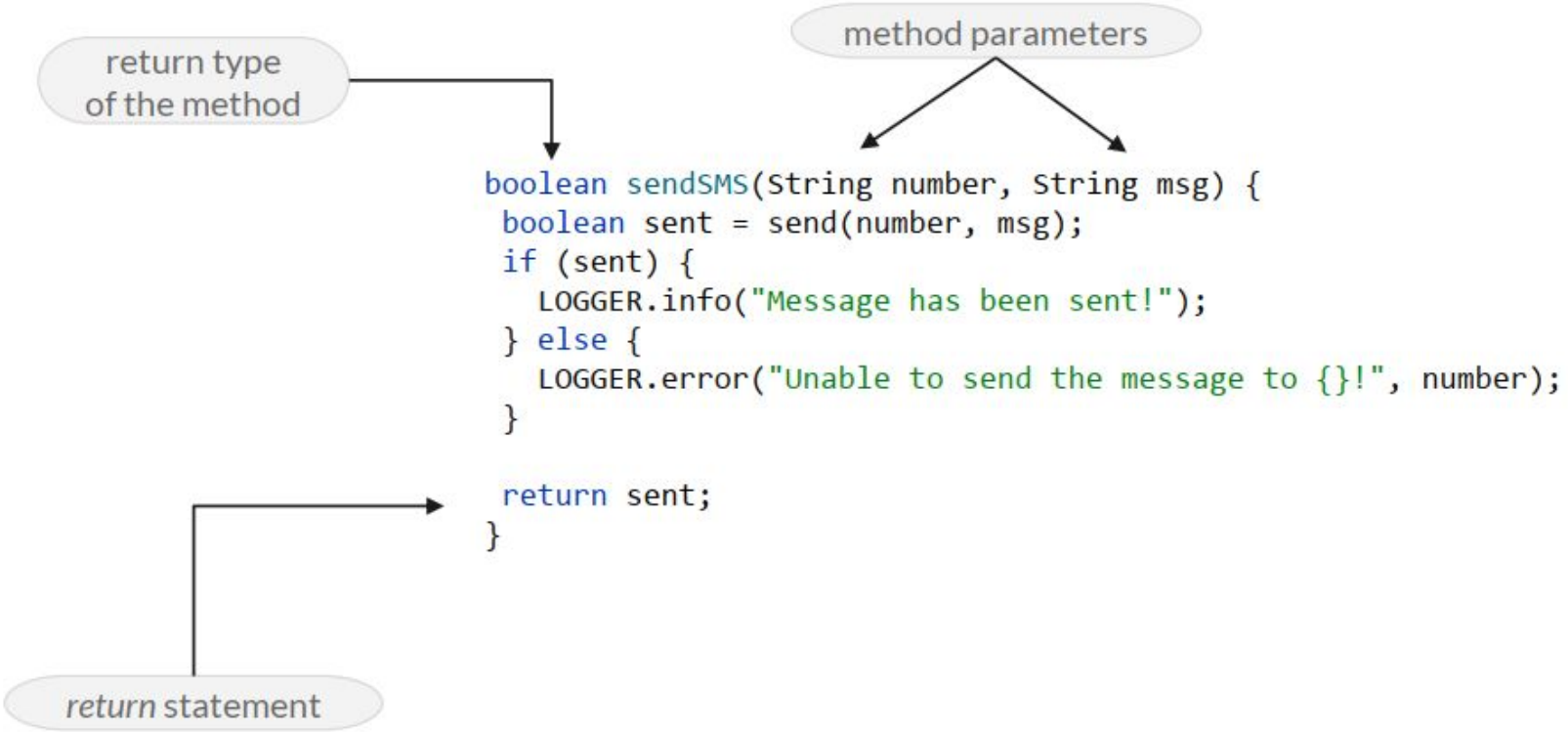
Creating and Using Methods

Basic Form of a Method

- A **method** is a block of code that performs a specific task.
- Methods represent *behaviors*.
- Methods perform actions; methods might return information about an object, or update an object's data.
- The method's code is defined in the class definition.

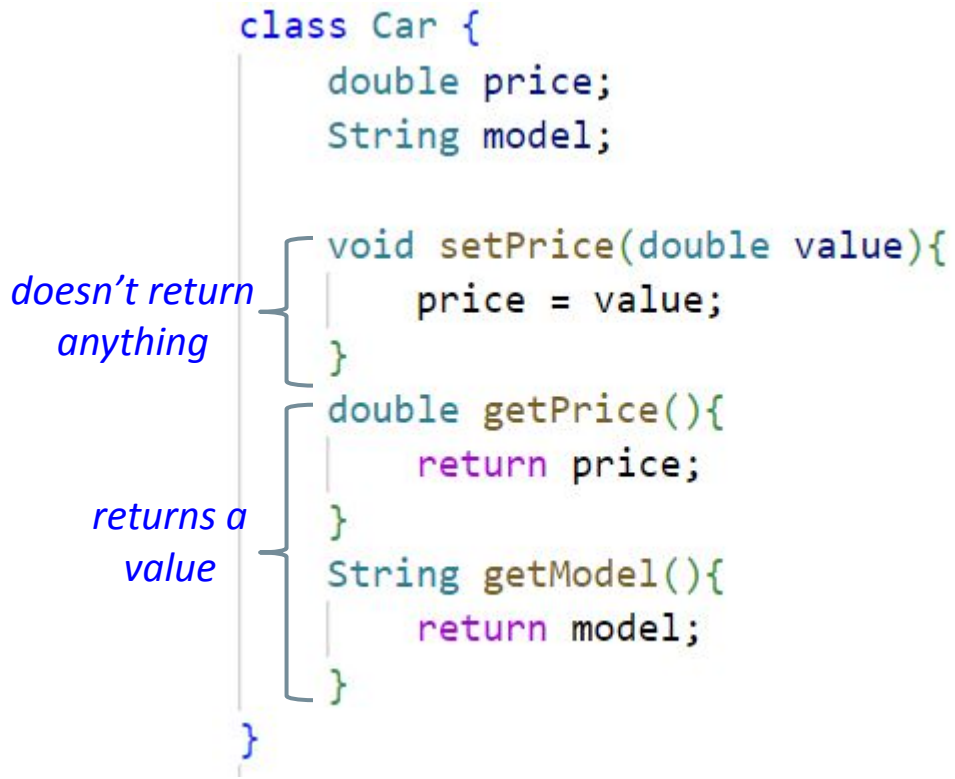
```
class Car {
    String color;
    String model;

    void setModel(String model){
        this.model = model;
    }
    String getModel(){
        return this.model;
    }
}
```

Basic Form of a Method



Return type of a method

- The **return type** of a method states the type of value that a method will return.
- A method may or may not return a value. One that doesn't return a value has a return type of void.



Return type of a method

- If a method doesn't return a value, you can't assign the result of that method to a variable.
- If a method returns a value, the calling method may or may not bother to store the returned value from a method in a variable.

```
class TestMethods {
    public static void main(String args[]) {
        Car p = new Car();

        //Because the method setWeight doesn't
        //return any value, this line won't compile.
        double price = p.setPrice(2000.0);

        //Method getPrice returns a double value,
        //but this value isn't assigned to any variable.
        p.getPrice();
    }
}
```

Method Parameters

- **Method parameters** are the variables that appear in the definition of a method and specify the type and number of values that a method can accept.
- You can pass multiple values to a method as input.
- **Theoretically**, no limit exists on the number of method parameters that can be defined by a method, but **practically** *it's not a good idea to define more than three method parameters*.

Method Parameters and Arguments

- **Method parameters** are the variables that appear in the definition of a method.

```
void divide(int x, double y) {
 // ...
}
```

parameters

- **Method arguments** are the actual values that are passed to a method while executing it.

```
Calculator calc = new Calculator();
double denominator = 2.0;

calc.divide(3, denominator);
```

arguments

Method Parameters

- A method may accept zero or multiple method arguments.

```
double calcAverage(int marks1, int marks2) {
    double avg = 0;
    avg = (marks1 + marks2)/2.0;
    return avg;
}

void printHello() {
    System.out.println(x: "Hello");
}
```

Method Parameters

- A method may accept variable arguments (**varargs**).

```
class Employee {
    public int daysOffWork(int... days) {
        int daysOff = 0;
        for (int i = 0; i < days.length; i++)
            daysOff += days[i];
        return daysOff;
    }
}
```

Method Parameters

- You can define only one variable argument in a parameter list, and it must be the last variable in the parameter list.

```
class Employee {
    public int daysOffWork(String... months, int... days) {
        int daysOff = 0;
        for (int i = 0; i < days.length; i++)
            daysOff += days[i];
        return daysOff;
    }
}
```

won't compile; you can't define multiple variables that can accept variable arguments.

Method Parameters

- You can define only one variable argument in a parameter list, and it must be the last variable in the parameter list.

```
class Employee {
    public int daysOffWork(int... days, String year) {
        int daysOff = 0;
        for (int i = 0; i < days.length; i++)
        daysOff += days[i];
        return daysOff;
    }
}
```

won't compile; if multiple parameters are defined, the variable argument must be the last in the list

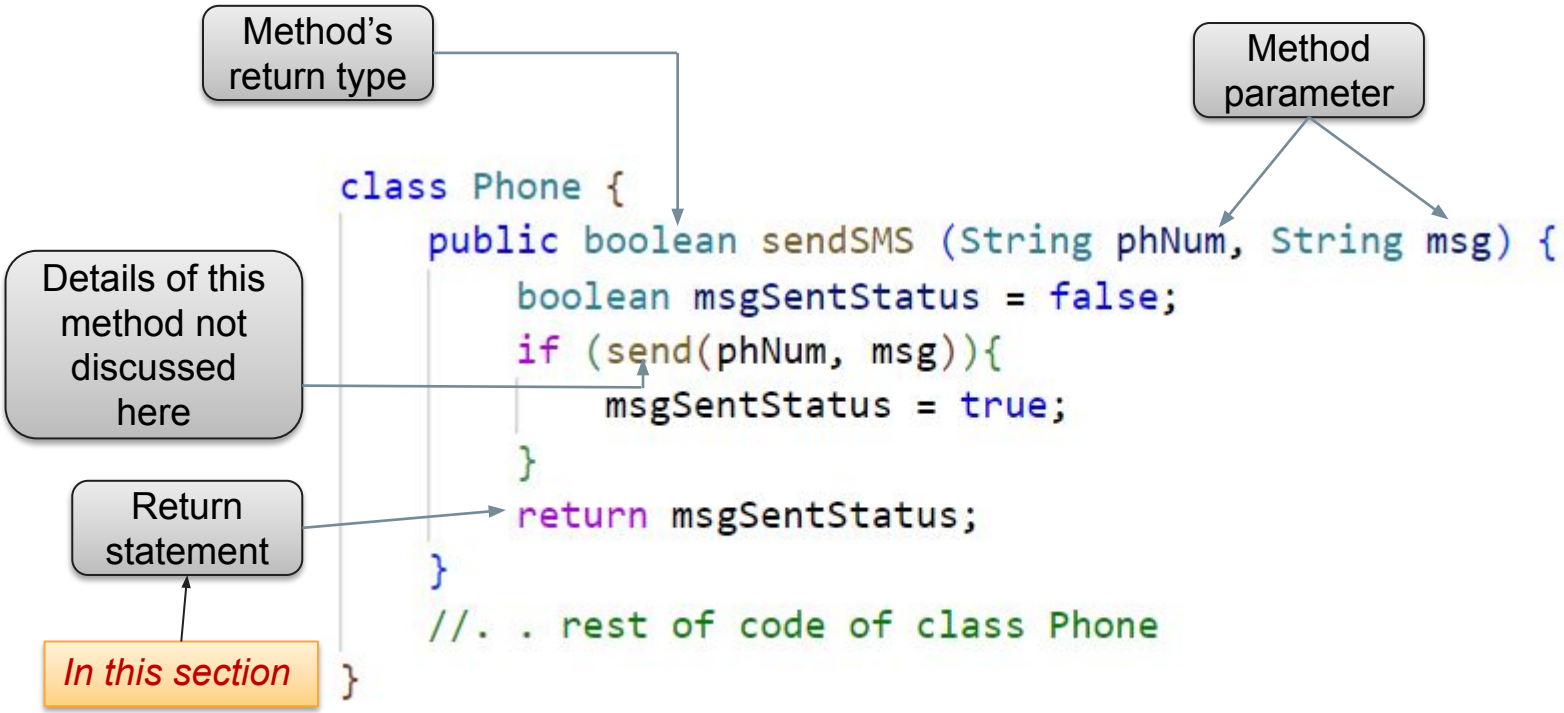
Rules to Remember About Method Parameters

- You can define multiple parameters for a method.
- The method parameter can be a primitive type or object.
- The method's parameters are separated by commas.
- Each method parameter is preceded by the name of its type. Each method parameter must have an explicit type declared with its name. You can't declare the type once and then list the parameters separated by commas, as you can for variables.

return Statement

- A **return** statement is used to exit from a method, with or without a value.
- For methods that define a return type, the **return** statement must be immediately followed by a return value.
- For methods that don’t return a value, the **return** statement can be used without a return value to exit a method.

return Statement



return Statement

- Method `calcAverage` returns a value of type double, using a `return` statement

```
double calcAverage(int marks1, int marks2) {
    double avg = 0;
    avg = (marks1 + marks2)/2.0;
    return avg;
}
```

- The methods that don't return a value (the return type is void) aren't required to define a `return` statement.

```
void setWeight(double val) {
    weight = val;
}
```

return Statement

- But you can use the `return` statement in a method even if it doesn't return a value. Usually this statement is used to define an early exit from a method.

```
void setWeight(double val) { // Method with return type void can use
    if (val < -1) return;    // return statement
    weight = val;
}
```

- Also, the `return` statement must be the last statement to execute in a method, if present.

```
void setWeight(double val) { //This code can't execute
    return; //due to the presence of the return statement before it
    weight = val;
}
```

return Statement

- The return statement need not be the last statement in a method, but it must be the last statement to execute in a method.

```
void setWeight(double val) {
    if (val < 0)
        return;
    else
        weight = val;
}
```

- The `return` statement isn't the last statement in this method. But it's the last statement to execute for method parameter values of less than zero.

Rules to Remember About a return Statement

- For a method that returns a value, the `return` statement must be followed immediately by a value.
- For a method that doesn't return a value (return type is void), the `return` statement must not be followed by a return value.
- If the compiler determines that a `return` statement isn't the last statement to execute in a method, the method will fail to compile.

Caller and Worker Methods

```
public static void main(String[] args) {
    Circle myCircle = new Circle();
    myCircle.radius = 10;
    myCircle.calculateArea();
    System.out.println(myCircle.getArea());
}
```

- The `main` method is referred to as the calling method because it is invoking or “calling” another method to do some work.
- The `calculateArea` method is referred to as the worker method because it does some work for the main method.

Code Without Methods

```
public static void main(String[] args){
    Shirt shirt01 = new Shirt();
    Shirt shirt02 = new Shirt();
    Shirt shirt03 = new Shirt();
    Shirt shirt04 = new Shirt();

    shirt01.description = "Sailor";
    shirt01.colorCode = 'B';
    shirt01.price = 30;

    shirt02.description = "Sweatshirt";
    shirt02.colorCode = 'G';
    shirt02.price = 25;

    shirt03.description = "Skull Tee";
    shirt03.colorCode = 'B';
    shirt03.price = 15;

    shirt04.description = "Tropical";
    shirt04.colorCode = 'R';
    shirt04.price = 20;
}
```

Better Code with Methods

```
public static void main(String[] args){
    Shirt shirt01 = new Shirt();
    Shirt shirt02 = new Shirt();
    Shirt shirt03 = new Shirt();
    Shirt shirt04 = new Shirt();

    shirt01.setFields("Sailor", 'B', 30);
    shirt02.setFields("Sweatshirt", 'G', 25);
    shirt03.setFields("Skull Tee", 'B', 15);
    shirt04.setFields("Tropical", 'R', 20);
}
```

```
public class Shirt {
    public String description;
    public char colorCode;
    public double price;

public void setFields(String desc,
                      char color,
                      double price){
    this.description = desc;
    this.colorCode = color;
    this.price = price;
}
```

Exercise #1 Operations with numbers

1. Create an `NumberMethods` class.
2. Implement and call the methods defined in the model `NumberMethods`.

   isEven – check if a number is even or odd.

   numberOfDigits2 – count the number of digits in a number that have the value 2.

   calculateMathOperation – take two numbers and a operation option and find a chosen math operation for two given numbers.

   sumDigits - compute the sum of the digits in a number.

```java
package methods;

public class NumberMethods {
 boolean isEven(int number) {
   return (number % 2 == 0);
 }

 int numberOfDigits2(int number) {
   int count = 0, digit;
   while (number != 0) {
     digit = number % 10;
     if (digit == 2) {
       count++;
     }
     number /= 10;
   }
   return count;
 }

 double calculateMathOperation(double number1, double number2, char operation) {
   double result = 0;
   switch (operation) {
     case '*': {
       result = number1 * number2;
       break;
     }
     case '-': {
       result = number1 - number2;
       break;
     }
     case '+': {
       result = number1 + number2;
       break;
     }
     case '/': {
       result = number1 / number2;
       break;
     }
   }
   return result;
 }

 int sumDigits(int number) {
   int sum = 0;
   while (number != 0) {
     sum += number % 10;
     number /= 10;
   }
   return sum;
 }

}
```

```java
package methods;

import java.util.Scanner;

public class NumberApplication {
 public static void main(String[] args) {
   Scanner scanner = new Scanner(System.in);
   NumberMethods numberMethods = new NumberMethods();
   double number1, number2;
   char operation;
   System.out.println("Input number");
   int number = scanner.nextInt();
   System.out.println("Statement: Number " + number + " is even is " +
numberMethods.isEven(number));
   System.out.println("Number " + number + " contains " +
numberMethods.numberOfDigits2(number) + " of digit 2");
   System.out.println("The sum of the digits is " +
numberMethods.sumDigits(number));
   System.out.println("Input the first number for calculate math operation");
   number1 = scanner.nextDouble();
   System.out.println("Input the second number for calculate math operation");
   number2 = scanner.nextDouble();
   System.out.println("Input the operation");
   operation = scanner.next().charAt(0);
   System.out.println(String.valueOf(number1) + operation +
String.valueOf(number2) + " = " +
numberMethods.calculateMathOperation(number1, number2, operation));
 }
}
```

Exercise #2 Operations with strings

1. Create an `StringMethods` class.
2. Implement and call the methods defined in the model `StringMethods`.

   countVowels – count all vowels in a string.

   countWords – count the number of words in a string.

   reverseString – print the reverse of the string by extracting and processing each character.

Exercise #2 Operations with strings

```
package com.amsoft.workshop.java.string;

import java.util.Arrays;

public class StringMethods {
   static int countVowels(String s) {
       String vow = "aeiou";
       int count = 0;
       char[] charArray = s.toCharArray();
       for (char ch : charArray) {
          if (vow.contains(String.valueOf(ch).toLowerCase()))
count++;

       }
       return count;
   }
```

```
static int countWords (String s){
       String[] wordArray = s.split("[, ;.!?:]");
       return (int) Arrays.stream(wordArray)
                              .filter(e->e.length()!=0).count();
   }
   static String reverseString(String s){
       return String.valueOf(new StringBuilder(s).reverse());
   }

}
```

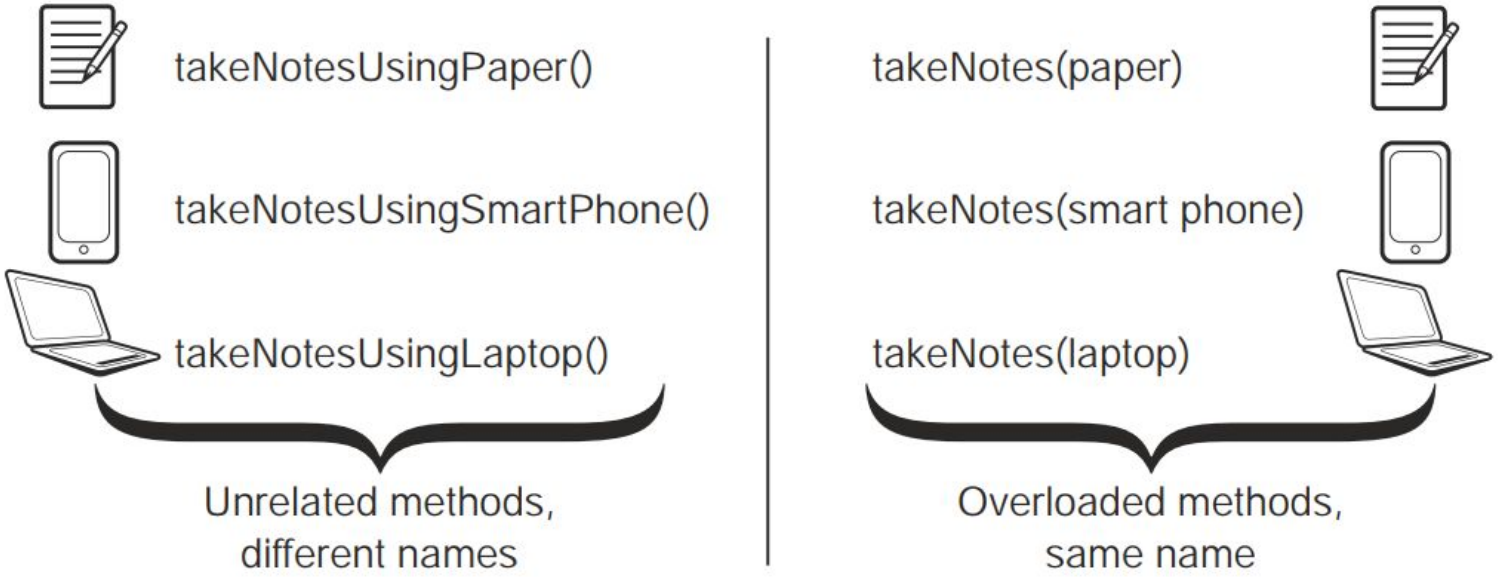
Exercise #2 Operations with strings

```
package com.amsoft.workshop.java.string;

import java.util.StringTokenizer;

public class StringApp {
   public static void main(String[] args) {
       System.out.println(StringMethods.countVowels("AlfaBEt"));
       System.out.println(StringMethods.countWords("Este o zi, o frumoasa zi de iarna!"));
       System.out.println(new StringTokenizer("Este o zi, o frumoasa zi de
iarna!").countTokens());
       System.out.println(StringMethods.reverseString("AbracadaBra"));

   }
}
```

Overloaded Methods

- *Overloaded methods* are methods with the same name but different method parameter lists.
- Overloaded methods make it easier to add methods with similar functionality that work with different sets of input values.

Overloaded Methods



- Overloaded methods are methods that are defined in the same class with the same name, but with different method argument lists.

Argument Lists

- Overloaded methods accept different lists of arguments.
- The argument lists can differ in terms of any of the following:
  - ✔ Change in the number of parameters that are accepted.
  - ✔ Change in the types of parameters that are accepted.
  - ✔ Change in the positions of the parameters that are accepted (based on parameter type, not variable names).

Examples of Overloaded Methods

```
double calcAverage(int marks1, double marks2) {
    return (marks1 + marks2) / 2.0;
}
double calcAverage(int marks1, int marks2, int marks3) {
    return (marks1 + marks2 + marks3) / 3.0;
}
double calcAverage(double marks1, int marks2) {
    return (marks1 + marks2) / 2.0;
}
double calcAverage(char marks1, char marks2) {
    return (marks1 + marks2) / 2.0;
}
```

byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double

Be Careful With The Arguments

The code will not compile if you try to execute the method using values that can be passed to both versions of the overloaded methods.

```
class MyClass {
    double calcAverage(double marks1, int marks2) {
        return (marks1 + marks2)/2.0;
    }
    double calcAverage(int marks1, double marks2) {
        return (marks1 + marks2)/2.0;
    }
    public static void main(String args[]) {
        MyClass myClass = new MyClass();
        myClass.calcAverage(2, 3);//Compiler can't determine
                                  //which overloaded method
                                  //calcAverage should be called
    }
}
```

Return Type

Methods can’t be defined as **overloaded methods** if they differ only in their *return types*.

```
double calcAverage(int marks1, int marks2) { //Return type is double
    return (marks1 + marks2)/2.0;
}

int calcAverage(int marks1, int marks2) { //Return type is int
    return (marks1 + marks2)/2;   // this method is not correctly overloaded
}
```

Access Level

Methods can’t be defined as **overloaded methods** if they differ only in their *access levels*.

```
public double calcAverage(int marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}

private double calcAverage(int marks1, int marks2) { //this method is not
    return (marks1 + marks2)/2.0;                     //correctly overloaded
}
```

Rules to Remember About Overloaded Methods

- Overloaded methods must have method parameters different from one another.
- Overloaded methods may or may not define a different return type.
- Overloaded methods may or may not define different access levels.
- Overloaded methods can’t be defined by only changing access modifiers or both.

Exercise #3 The perimeter of a geometric shape

Create a Java class to calculate the perimeter of a geometric shape (trapezoid, triangle, rectangle, square). Determining the methods to be implemented:

```
int calculatePerimeter(int side1, int side2, int side3, int side4)
int calculatePerimeter(int side1, int side2, int side3)
int calculatePerimeter(int length, int width)
int calculatePerimeter(int sideLength)
```

```java
package methods;

public class ShapeCalculator {
 int calculatePerimeter(int side1, int side2, int side3, int side4) {
   return side1 + side2 + side3 + side4;
 }

 int calculatePerimeter(int side1, int side2, int side3) {
   return side1 + side2 + side3;
 }

 int calculatePerimeter(int length, int width) {
   return 2 * (length + width);
 }

 int calculatePerimeter(int sideLength) {
   return 4 * sideLength;
 }
}
```

```java
package methods;
import java.util.Scanner;
public class ShapeApplication {
 public static void main(String[] args) {
   int perimeter=0;
   ShapeCalculator shapeCalculator=new ShapeCalculator();
   Scanner scanner=new Scanner(System.in);
   System.out.println("Input what kind of shape is it (1-square, 2-rectangle, 3-triangle,
4-trapezoid)");
   int selectValue=scanner.nextInt();
   switch(selectValue){
     case 1:{
       System.out.print("Input side length ");
       int sideLength=scanner.nextInt();
       perimeter=shapeCalculator.calculatePerimeter(sideLength);
       break;
     }
     case 2:{
       System.out.print("Input length ");
       int length=scanner.nextInt();
       System.out.print("Input width ");
       int width=scanner.nextInt();
       perimeter=shapeCalculator.calculatePerimeter(length,width);
       break;
     }
     case 3:{
       System.out.print("Input side1 ");
       int side1=scanner.nextInt();
       System.out.print("Input side2 ");
       int side2=scanner.nextInt();
       System.out.print("Input side3 ");
       int side3=scanner.nextInt();
       perimeter=shapeCalculator.calculatePerimeter(side1,side2,side3);
       break;
     }
     case 4:{
       System.out.print("Input side1 ");
       int side1=scanner.nextInt();
       System.out.print("Input side2 ");
       int side2=scanner.nextInt();
       System.out.print("Input side3 ");
       int side3=scanner.nextInt();
       System.out.print("Input side4 ");
       int side4=scanner.nextInt();
       perimeter=shapeCalculator.calculatePerimeter(side1,side2,side3,side4);
       break;
     }
     default: {
       System.out.println("The program can't calculate perimeter for this kind of geometric shape");
     }
   }
   if (perimeter!=0) {System.out.println("Perimeter of shape is "+perimeter);}
 }
}
```

Passing Objects and Primitives to Methods

Passing Primitives to Methods

The value of a primitive data type is copied and passed to a method.

The variable whose value was copied doesn't change.

```
class Employee {
    int age;

    void modifyVal(int a) {
        a = a + 1;
        System.out.println(a);
    }
}
```

```
class Office {
    public static void main(String args[]) {
        Employee e = new Employee();
        System.out.println(e.age);
        e.modifyVal(e.age);
        System.out.println(e.age);
    }
}
```

Output:
0 1 0

Passing Primitives to Methods

What happens if the definition of the class Employee is modified as follows.

```
class Employee {
    int age;

    void modifyVal(int age) {
        age = age + 1;
        System.out.println(age);
    }
}
```

```
class Office {
    public static void main(String args[]) {
        Employee e = new Employee();
        System.out.println(e.age);
        e.modifyVal(e.age);
        System.out.println(e.age);
    }
}
```

Output:
0 1 0

When you pass a primitive variable to a method, its value remains the same after the execution of the method. The value doesn't change, regardless of whether the method reassigns the primitive to another variable or modifies it.

Passing Object References To Methods

There are two main cases:

- When a method reassigns the object reference passed to it to another variable.
- When a method modifies the state of the object reference passed to it.

When Methods Reassign The Object References Passed To Them

When you pass an object reference to a method, the method can assign it to another variable.

In this case the state of the object, which was passed on to the method, remains intact.

```
class Person {
    private String name;
    Person(String newName) {
        name = newName;
    }
    public String getName() {
        return name;
    }
    public void setName(String val) {
        name = val;
    }
}
```

When Methods Reassign The Object References Passed To Them

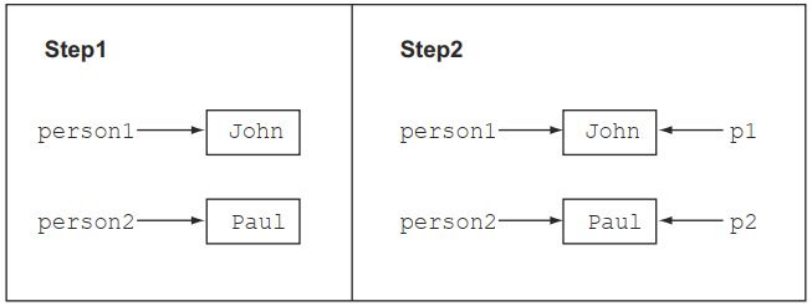
```
class Test {
    public static void swap(Person p1, Person p2) {
        Person temp = p1;
        p1 = p2;
        p2 = temp;
    }
    public static void main(String args[]) {
        Person person1 = new Person("John");
        Person person2 = new Person("Paul");
        System.out.println(person1.getName()
                           + ":" + person2.getName());
        swap(person1, person2);
        System.out.println(person1.getName()
                           + ":" + person2.getName());
    }
}
```

Creates object

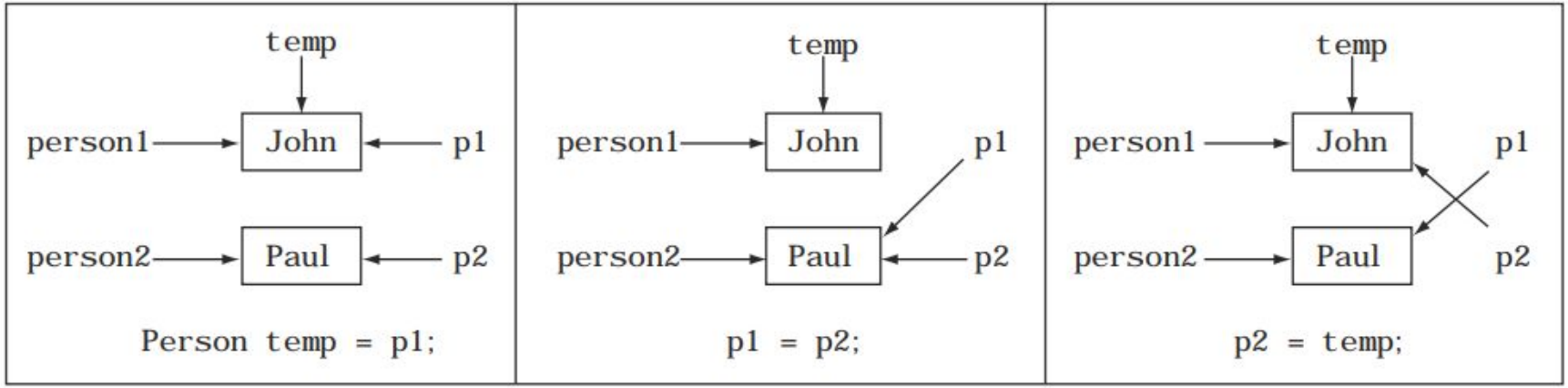
Prints John:Paul

Executes method swap

Prints John:Paul



When Methods Reassign The Object References Passed To Them



Output:
John:Paul
John:Paul

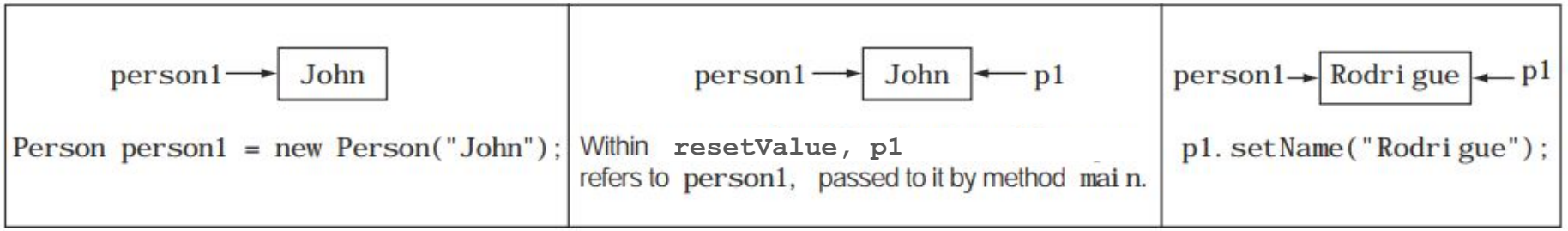
When Methods Modify The State Of The Object References Passed To Them

```java
class Person {
    private String name;
    Person(String newName) {
        name = newName;
    }
    public String getName() {
        return name;
    }
    public void setName(String val) {
        name = val;
    }
}
```

```java
class Test {
    public static void resetValue(Person p1) {
        p1.setName("Rodrigue");
    }
    public static void main(String args[]) {
        Person person1 = new Person("John");
        System.out.println(person1.getName());
        resetValue(person1);
        System.out.println(person1.getName());
    }
}
```

Output:
John
Rodrigue

When Methods Modify The State Of The Object References Passed To Them



Scope of Variables

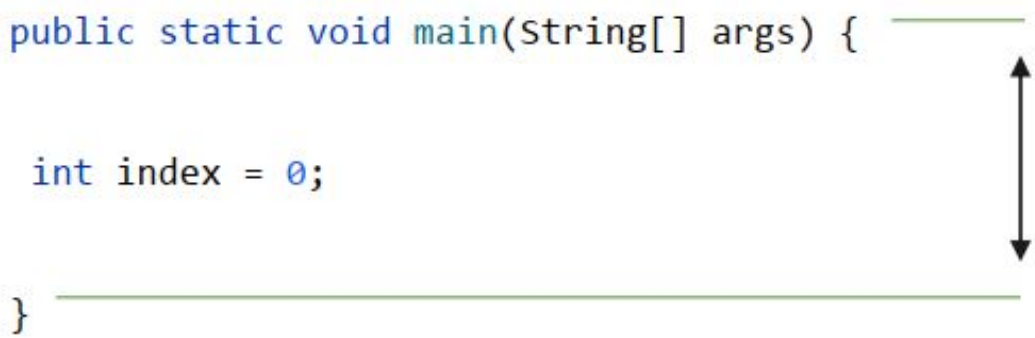
The *scope of a variable* specifies its life span and its visibility.

Here are the available scopes of variables:

- **Local variables** (also known as method-local variables).
- **Method parameters** (also known as method arguments).
- **Instance variables** (also known as attributes, fields, and nonstatic variables).
- **Class variables** (also known as static variables).

Scope of Variables

As a rule of a thumb, the scope of a variable ends when the brackets of the block of code it's defined in get closed.



Local Variables

```
public class TestApp {
    public static void main(String[] args)
    {
        m1(); // main() method calling m1().
    }
    public static void m1()
    {
        m2(); // m1() method calling m2().
    }
    public static void m2()
    {
        m3(); // m2() method calling m3().
    }
    public static void m3()
    {
        System.out.println(1); // print value 1
    }
}
```

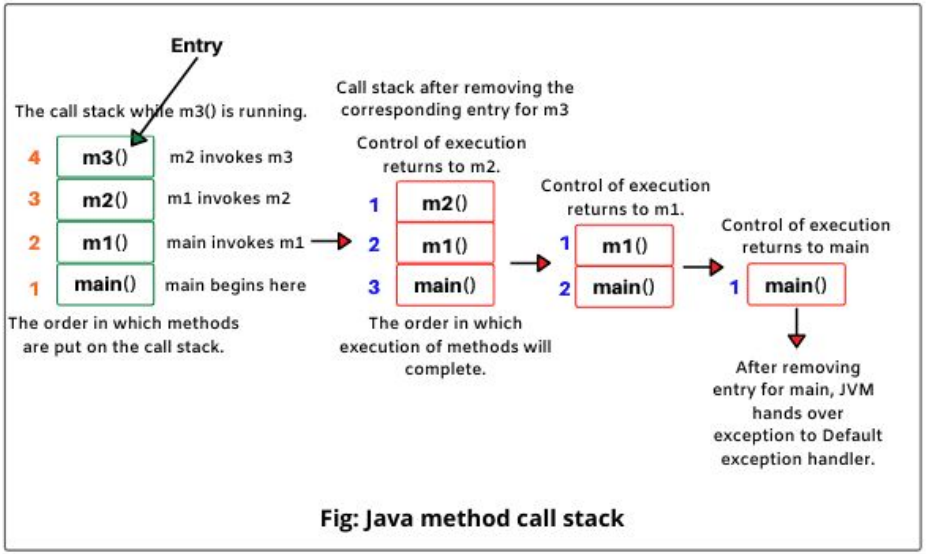
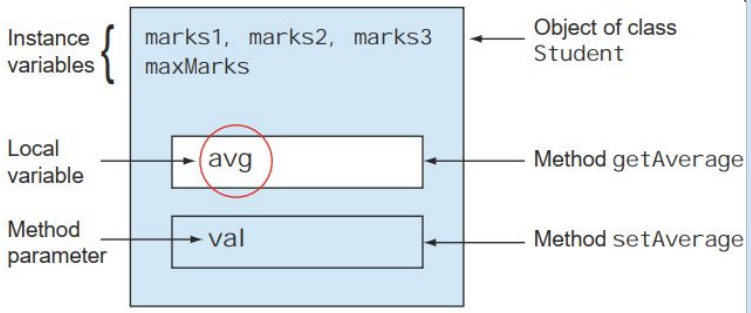


Fig: Java method call stack

Local Variables

Local variables are defined within a method. Typically, you'd use local variables to store the intermediate results of a calculation.

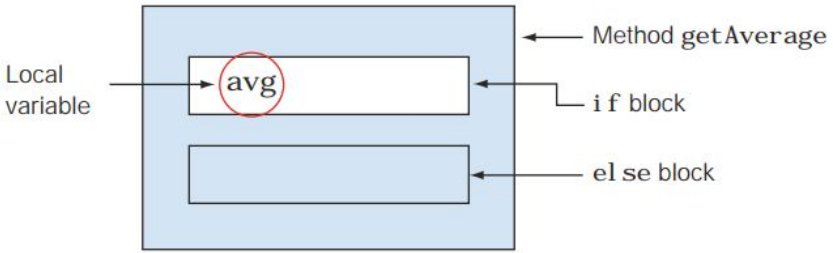
```
class Student {
    private double marks1, marks2, marks3; //Instance variables
    private double maxMarks = 100;
    public double getAverage() {
        double avg = 0; //Local variable avg
        avg = ((marks1 + marks2 + marks3) / (maxMarks*3)) * 100;
        return avg;
    }
    public void setAverage(double val) {
        avg = val;//This code won't compile because avg
                  //is inaccessible outside the method getAverage.
    }
}
```



Local Variables

Local variables are defined within a method. Typically, you'd use local variables to store the intermediate results of a calculation.

```
public double getAverage() {
    if (maxMarks > 0) {
        double avg = 0;//Variable avg is local to if block
        avg = (marks1 + marks2 + marks3)/(maxMarks*3) * 100;
        return avg;
    }
    else {
        //// variable avg can't be accessed because it's local to the if block
        avg = 0;
        return avg;
    }
}
```



Local Variables

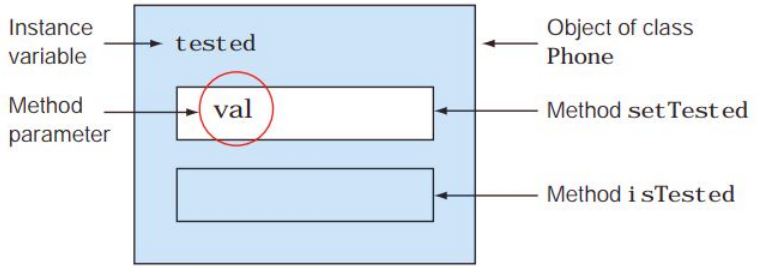
- The scope of a local variable depends on the location of its declaration within a method.
- The scope of local variables defined within a loop, if-else, or switch construct or within a code block (marked with {}) is limited to these constructs.
- Local variables defined outside any of these constructs are accessible across the complete method.

Method Parameters

The variables that accept values in a method signature are called **method parameters**. They're accessible only in the method that defines them.

```
class Phone {
    private boolean tested;
    public void setTested(boolean val) {
        //Method parameter val is accessible only in method setTested
        tested = val;
    }
    public boolean isTested() {
        val = false;//Variable val can't be accessed in method isTested
        return tested;
    }
}
```

The scope of a method parameter may be as long as that of a local variable or longer, but it can never be shorter.



Instance Variables

Instance is another name for an object. Hence, an **instance variable** is available for the life of an object.

An instance variable is declared within a class, outside all the methods.

```
class Phone {
    private boolean tested;//Instance variable tested
    public void setTested(boolean val) {
        //Variable tested is accessible in method setTested
        tested = val;
    }
    public boolean isTested() {
        //Variable tested is also accessible in method isTested
        return tested;
    }
}
```

Class Variables

A **class variable** is defined by using the keyword **`static`**.

A **class variable** belongs to a class, not to individual objects of the class.

A **class variable** is shared across all objects - objects don't have a separate copy of the class variables.

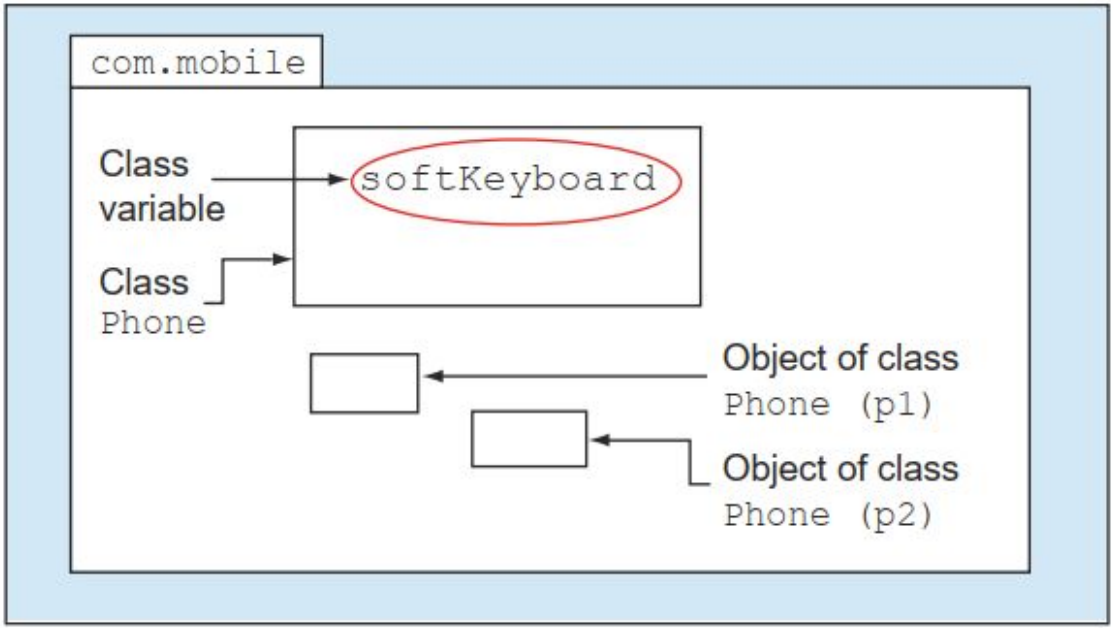
A **class variable** can be accessed by using the name of the class in which it's defined.

```
class Phone {
    //Class variable softKey
    static boolean softKeyboard = true;
}
```

Class Variables

```
class TestPhone {
    public static void main(String[] args) {
        Phone.softKeyboard = false;
        Phone p1 = new Phone();
        Phone p2 = new Phone();

        System.out.println(p1.softKeyboard); //false
        System.out.println(p2.softKeyboard); //false

        // the change will be reflected in all the objects of this class
        p1.softKeyboard = true;
        System.out.println(p1.softKeyboard); //true
        System.out.println(p2.softKeyboard); //true
        System.out.println(Phone.softKeyboard);//true
    }
}
```

Class Variables



The variable `softKeyboard` *is accessible even without the existence of any Phone instance*

Comparing the Use of Variables in Different Scopes

- **Local variables** are defined within a method and are normally used to store the intermediate results of a calculation.
- **Method parameters** are used to pass values to a method.
- **Instance variables** are used to store the state of an object. These are the values that need to be accessed by multiple methods.
- **Class variables** are used to store values that should be shared by all the objects of a class.

Access modifiers

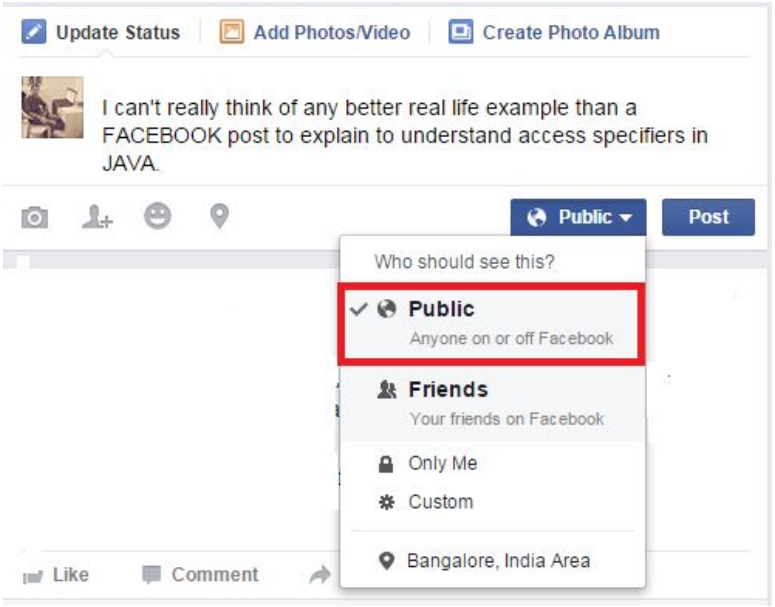
What are Access Modifiers?

Public: Anyone can see your profile

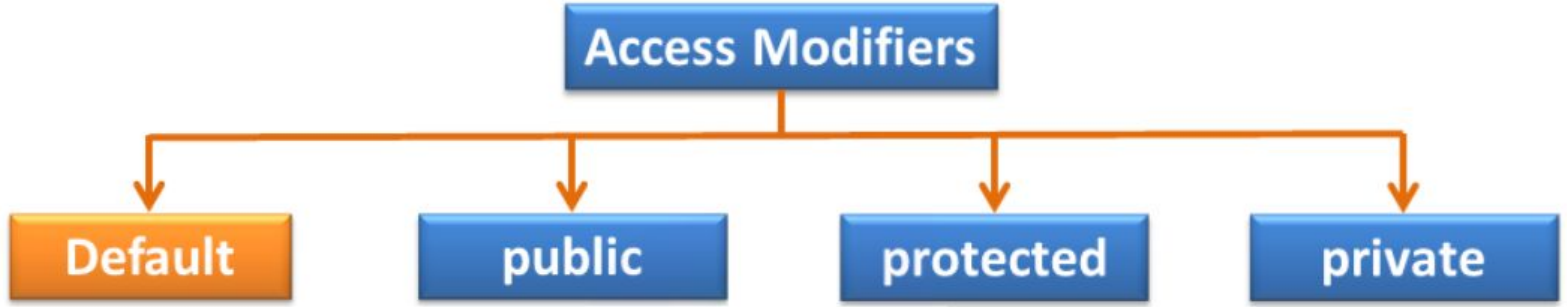
Protected : Only your friends

Default: Only specific user group

Private : Only me



Access Modifiers



Default	public	protected	private
Visible to the package, the default. No modifiers are needed.	Visible to the world	Visible to the package and all subclasses	Visible to the class only

Access Modifiers

Let us see which all members of Java can be assigned with the access modifiers:

Members of JAVA	Private	Default	Protected	Public
Class	No	Yes	No	Yes
Variable	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
interface	No	Yes	No	Yes
Initializer Block	NOT ALLOWED			

Access modifiers: public

- The public access modifier is specified using the keyword `public`.
- A member that is declared as `public` is accessible within the class and all of its sub-classes even if those sub-classes are present in other packages.
- Also, public members are accessible outside of the class in any package through its instance.

```
package pkg1;

public class Alpha { // public class
    public String publicA = "public variable is accessible"; // public variable

    public void display() { // public method
        System.out.println("Within the class Alpha " + this.publicA);
    }
}
```

Access modifiers: public

```
package pkg2;

public class Pkg2App {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        alpha.display();
        System.out.println("Outside of Alpha " + alpha.publicA);

        Gamma gamma = new Gamma();
        gamma.display();
        System.out.println("Outside of Gamma " + gamma.publicA);
    }
}
```

```
Within the class Alpha public variable is accessible
Outside of Alpha public variable is accessible
Within the class Gamma public variable is accessible
Outside of Gamma public variable is accessible
```

Access modifiers: public

Classes that can access a public class and its members

Access modifiers: protected

The members of a class defined using the `protected` access modifier are accessible to:

- Classes defined in the same package;
- All derived classes, even if they're defined in separate packages.

```
package pkg1;

public class Alpha { // public class
    protected String protectedB = "protected variable is accessible"; // protected variable

    public void display() { // public method
        System.out.println("Within the class Alpha " + this.protectedB);
    }
}
```

Access modifiers: protected

```
package pkg1;

public class Pkg1App {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        alpha.display();
        System.out.println("Outside of Alpha " + alpha.protectedB);

        Beta beta = new Beta();
        beta.display();
        System.out.println("Outside of Beta " + gamma.protectedB);
    }
}
```

```
Within the class Alpha protected variable is accessible
Outside of Alpha protected variable is accessible
Within the class Beta protected variable is accessible
Outside of Beta protected variable is accessible
```

Access modifiers: protected

```java
package pkg2;

public class Pkg2App {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        alpha.display();

        Gamma gamma = new Gamma();
        gamma.display();
//          System.out.println("Outside of Gamma " + gamma.protectedB);
    }
}
```

```
Within the class Alpha protected variable is accessible
Within the class Gamma protected variable is accessible
```

```
java: protectedB has protected access in com.workshop.oop.lesson2.accessmodifiers.pkg1.Alpha
```

Access modifiers: protected

Classes that can access protected members

Access modifiers: default/package access

- The members of a class defined without using any explicit access modifier are defined with `package accessibility` (also called default accessibility).
- The members with package access are **only** accessible to classes defined in the same package.

```
package pkg1;

public class Alpha { // public class
    String noModifierC = "default variable is accessible"; // variable with default access

    public void display() { // public method
        System.out.println("Within the class Alpha " + this.noModifierC);
    }
}
```

Access modifiers: default/package access

```java
package pkg1;

public class Pkg1App {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        alpha.display();
        System.out.println("Outside of Alpha " + alpha.noModifierC);

        Beta beta = new Beta();
        beta.display();
        System.out.println("Outside of Beta " + beta.noModifierC);

        Gamma gamma = new Gamma();
        gamma.display();
        // System.out.println("Outside of Alpha " + gamma.noModifierC);
    }
}
```

```
Within the class Alpha default variable is accessible
Outside of Alpha default variable is accessible
Within the class Beta default variable is accessible
```

```
java: noModifierC is not public in com.workshop.oop.lesson2.accessmodifiers.pkg1.Alpha; cannot be accessed from outside package
```

Access modifiers: default/package access

The classes that can access members with default (package) access

Access modifiers: private

- The `private` access modifier is the most restrictive access modifier.
- The members of a class defined using the `private` access modifier are accessible only to themselves.
- `private` members are not accessible outside the class in which they're defined.

```
package pkg1;

public class Alpha { // public class
    private String privateD = "private variable is accessible"; // private variable

    public void display() { // public method
        // only Alpha can access its own private variable privateD
        System.out.println("Within the class Alpha " + this.privateD);
    }
}
```

Access modifiers: private

```
package pkg1;

public class Beta extends Alpha {

    public void display() {
        System.out.println("Within the class Beta " + this.privateD);
    }

}
```

```
package pkg1;

public class Pkg1App {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        alpha.display();
        System.out.println("Outside of Alpha " + alpha.privateD);
    }
}
```

```
Within the class Alpha private variable is accessible
```

Access modifiers: private

No classes can access `private` members of another class

Access modifiers

Non-access modifiers

Non-access modifiers

- Access modifiers control the accessibility of your class and its members outside the class and the package.

- Non-access modifiers change the default behavior of a Java class and its members.

  - `abstract`
  - `final`
  - `static`

Non-access modifiers

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For **attributes and methods**, you can use the one of the following:

Non-access modifiers

For **attributes and methods**, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **`abstract void run();`**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
`transient`	Attributes and methods are skipped when serializing the object containing them
`synchronized`	Methods can only be accessed by one thread at a time
`volatile`	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Non-access modifiers: abstract

- When added to the definition of a class or method, the `abstract` modifier changes its default behavior.
- Because it is a non-access modifier, `abstract` doesn't change the accessibility of a class or method.

Different contexts where abstract can be used in Java:

- Abstract classes
- Abstract methods

abstract class

- When the `abstract` keyword is prefixed to the definition of a concrete class, it changes it to an *abstract class*, even if the class doesn't define any abstract methods.
- The *abstract class* in Java cannot be instantiated (we cannot create objects of abstract classes).
- An *abstract class* can have both the regular methods and abstract methods.

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

abstract method

- A method that doesn't have its body is known as an abstract method.
- If a class contains an abstract method, then the class should be declared abstract.
- An abstract method is implemented by a derived class.

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Non-access modifiers: final

- The `final` keyword is used to denote constants.
- It can be used with *variables*, *methods*, and *classes*.
- Once any entity (variable, method or class) is declared `final`, it can be assigned only once. That is,
  - the final variable cannot be reinitialized with another value
  - the final method cannot be overridden
  - the final class cannot be extended

Non-access modifiers: final

```
public class Main {
  final int x = 10;
  final double PI = 3.14;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 50; // will generate an error: cannot assign a value to a final variable
    myObj.PI = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }

  }
```

final variable

- We cannot change the value of a `final` variable.

```
class Main {
  public static void main(String[] args) {

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

final method

- The `final` method cannot be overridden by the child class.

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

```
display() in Main cannot override display() in FinalDemo
 public final void display() {
                   ^
 overridden method is final
```

final class

- The `final` class cannot be inherited by another class.

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

Non-access modifiers: static

The non-access modifier `static` can be used with variables, methods, blocks and nested classes.

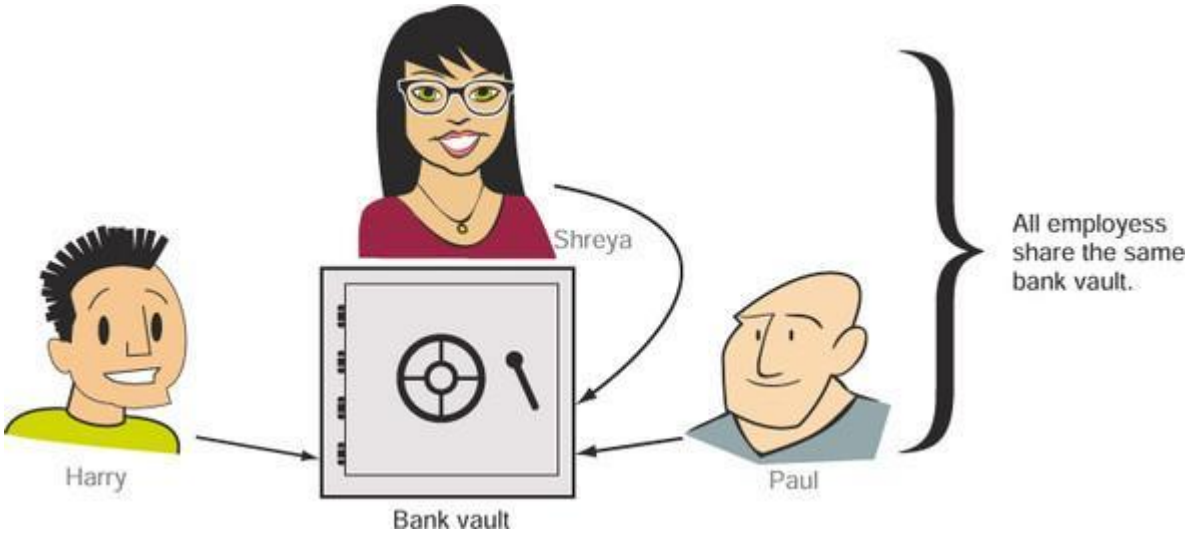
The `static` keyword belongs to the class than an instance of the class.

static variables

- `static` variables belong to a class.
- `static` attributes exist independently of any instances of a class and may be accessed even when no instances of the class have been created.
- They’re common to all instances of a class and aren’t unique to any instance of a class.

static variables

- You can compare a static variable with a shared variable.
- A static variable is shared by all the instances of a class.



static variables

Definition of the class `Emp` with a static variable `bankVault` and non-static variable `name`.

```
class Emp {
    String name;

    // we want this value to be shared by all the objects of class Emp
    static int bankVault;
}
```

static variables

```
class TestEmp {
    public static void main(String[] args) {
        Emp emp1 = new Emp(); //References variables emp1 and emp2 refer
        Emp emp2 = new Emp(); //to separate objects of class
        emp1.bankVault = 10; //is assigned a value 10
        emp2.bankVault = 20; //is assigned a value 20
        System.out.println(emp1.bankVault); //This will print 20
        System.out.println(emp2.bankVault); //This will print 20
        System.out.println(Emp.bankVault); //This will print 20
    }
}
```

***Note**: Using an object reference variable to access static members is not recommended because static members belong to the class, not individual objects.*

*The preferred way to access them is by **using the class name.***

static variables as constants

The static and final nonaccess modifiers can be used together to define **constants**.

```
class Emp {
    // constant MIN_AGE
    public static final int MIN_AGE = 20;

    // constant MAX_AGE
    static final int MAX_AGE = 70;
}
```

static methods

- `static` methods aren't associated with objects and can't use any of the instance variables of a class.
- You can define `static` methods to access or manipulate static variables.

```
class Emp {
    String name;
    static int bankVault;

    // static method getBankVaultValue returns the value of static variable bankVault
    static int getBankVaultValue() {
        return bankVault;
    }
}
```

***Note**: It's a common practice to use static methods to define utility methods.*

static methods

```
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[ ] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method
  }
}
```

What can a static method access?

- Neither **`static` methods nor static variables** can access the non-static variables and methods of a class.
- But the reverse is true: **non-static variables and methods** can access static variables and methods because the static members of a class exist even if no instances of the class exist.

```
class MyClass {
    static int x = count(); // compilation error
    int count() {
        return 10;
    }
}
```

***Note**: static methods and variables can't access the instance members of a class.*

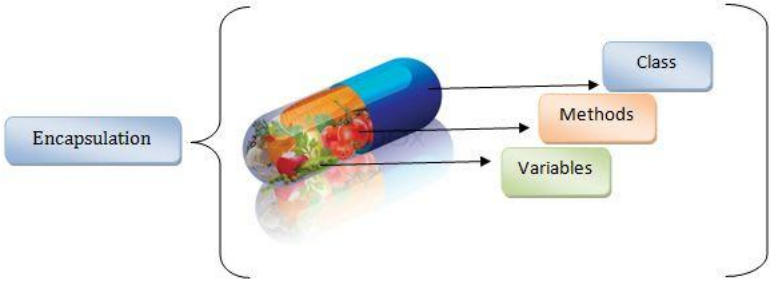
Access capabilities of static and non-static members

Member type	Can access `static` attribute or method?	Can access non-`static` attribute or method?
`static`	Yes	No
Non-`static`	Yes	Yes

Encapsulation

What is Encapsulation?

- Encapsulation is defined as the wrapping up of data under a single unit.
- A well-encapsulated object doesn't expose its internal parts to the outside world. It defines a set of methods that enables the users of the class to interact with it.
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.



Need for encapsulation

```
class Phone {
    //instance variables that store the state of an object of Phone
    String model;
    String company;
    double weight;
    void makeCall(String number) { }
    void receiveCall() { }
    }
    class Home {
        public static void main() {
            Phone ph = new Phone();
            ph.weight = -12.23; // assign a negative weight to Phone
        }
}
```

- To prevent an external object from performing dangerous operations.
- To hide implementation details, so that the implementation can change a second time without impacting other objects.

Apply encapsulation

- The method `setWeight` doesn't assign the value passed to it as a method parameter to the instance variable `weight` if it's a negative value or a value greater than 1,000.
- This behavior is known as **exposing object functionality using public methods**.

```
class Phone {
    private double weight;

    public void setWeight(double val) {
        if (val >= 0 && val <= 1000) { // negative and weight over 1000 not allowed
            weight =val;
        }
    }
    public double getWeight() {
        return weight;
    }
}
```

Apply encapsulation

```
class Home {
    public static void main(String[] args) {
        Phone ph = new Phone();

        ph.setWeight(-12.23); // assign a negative weight
        System.out.println(ph.getWeight()); // prints 0.0

        ph.setWeight(77712.23); // assign weight > 1,000
        System.out.println(ph.getWeight()); // prints 0.0

        ph.setWeight(12.23); // Assign weight in allowed range
        System.out.println(ph.getWeight()); // prints 12.23
    }
}
```

Get and Set methods

- It is recommended that all fields of a class should be private, and those that need to be accessed should have public methods for setting and getting their values.
- To make a **private data field** accessible, provide a **getter** method to return its value. A getter method has the following signature:

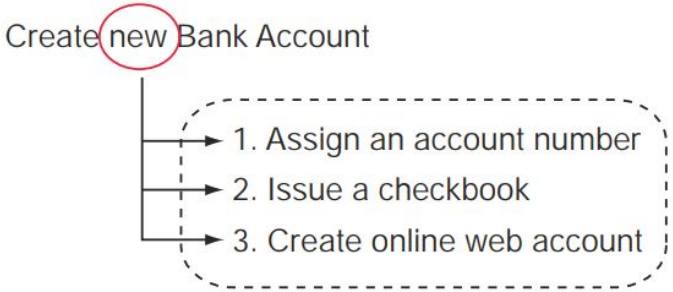
```
public returnType getPropertyName()
```

- To enable a **private data field** to be updated, provide a **setter** method to set a new value. A setter method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Constructors of a Class

Constructors of a Class



The series of steps that may be executed when you create a new bank account.

- **Constructors** are special methods that *create* and *return* an object of the class in which they're defined.
- Constructors have the same name as the class in which they're defined, and they don't specify a return type – not even `void`.

Constructors of a Class

A constructor method is a special method that is invoked **when you create an object instance**.

- It is called by using the new keyword.
- Its purpose is to instantiate an object of the class and store the reference in the reference variable.

```
Book myBook = new Book();
```

Constructor method is called

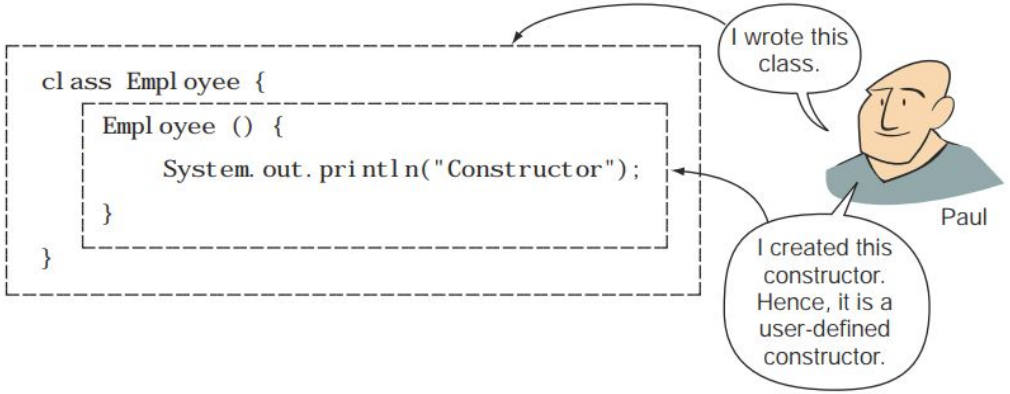
- It has a unique method signature.

```
<modifier> ClassName()
```

User-defined constructors

The author of a class has full control over the definition of the class. An author may or may not define a constructor in a class. If the author defines a constructor in a class, it's known as a **user-defined constructor**.

Here the word *user* doesn't refer to another person or class that uses this class but instead refers to the person who created the class. It's called "*user-defined*" because it's not created by the Java compiler.



Using a user-defined constructor

Constructor is used to assign default values to an instance variable of your class.

```java
class Employee {
    //Instance variables
    String name;
    int age;
    Employee() {
        age = 20; //Initialize age
        System.out.println("Constructor");
    }
}
```

```java
class Office {
    public static void main(String args[]) {
        Employee emp = new Employee();

        //Access and print the value of variable age
        System.out.println(emp.age);
    }
}
```

Output: Constructor
20

Using a user-defined constructor

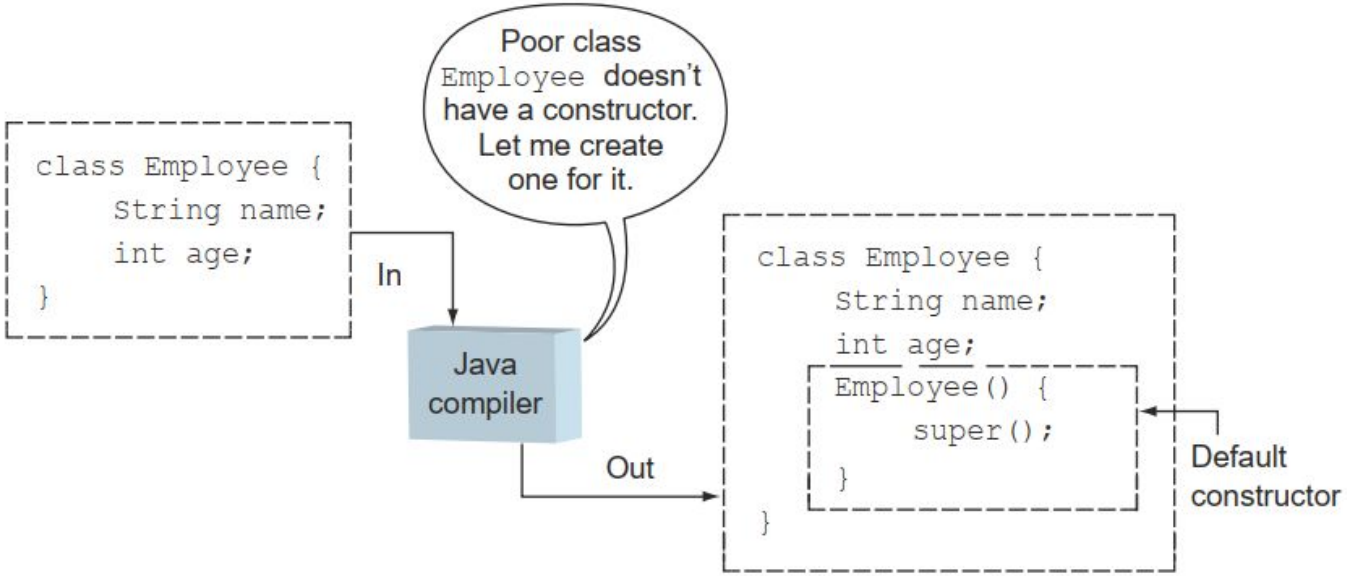
Because a constructor is like a method, you can also pass method parameters to it.

```java
class Employee {
    String name;
    int age;
    Employee(int newAge, String newName) {
        name = newName;
        age = newAge;
        System.out.println("Constructor");
    }
}
```

```java
class Office {
    public static void main(String args[]) {
        Employee emp = new Employee(45, "Jhon");
    }
}
```

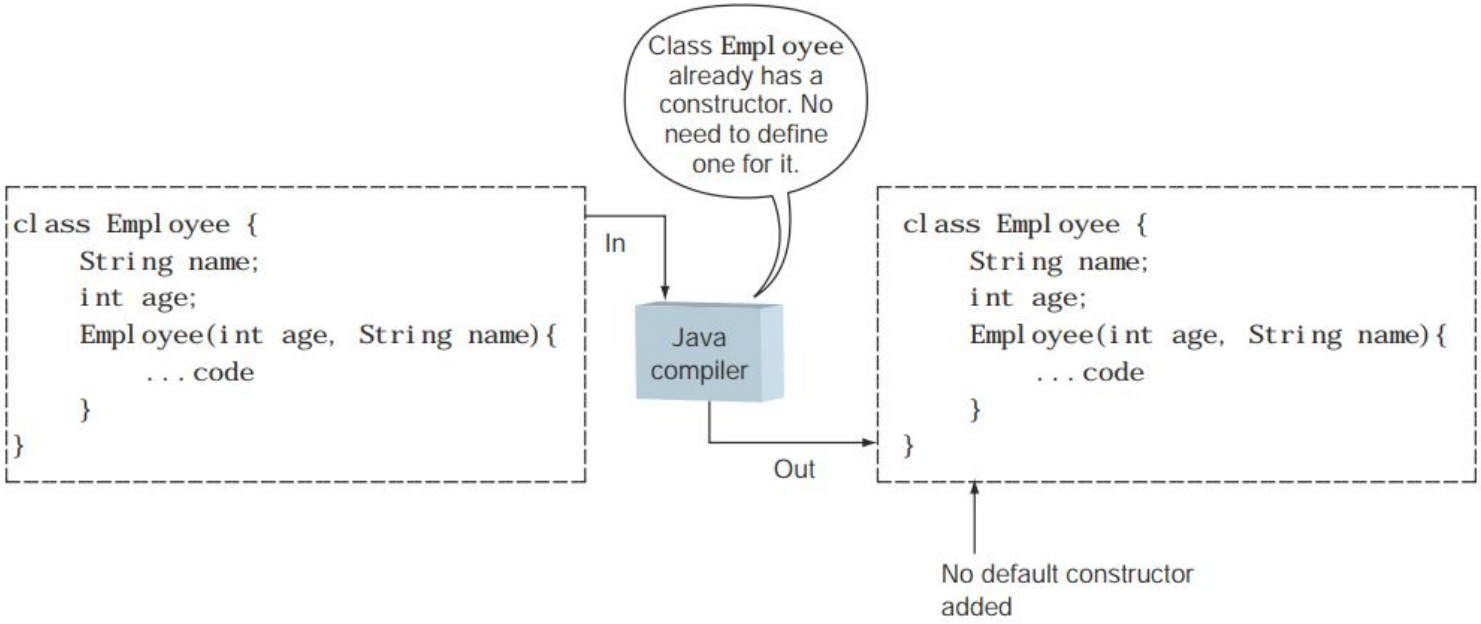
Default constructor

In the absence of a user-defined constructor, Java inserts a **default constructor**.



Default constructor

Java defines a default constructor if and only if you don’t define a constructor.



Default constructor

Java defines a default constructor if and only if you don’t define a constructor.

```
class Employee {
    String name;
    int age;
    Employee(int newAge, String newName) { // user-defined constructor
        name = newName;
        age = newAge;
        System.out.println("User defined Constructor");
    }
}
class Office {
    public static void main(String args[]) {
        Employee emp = new Employee(); // won’t compile
    }
}
```

Overloaded constructors

In the same way in which you can overload methods in a class, you can also overload the constructors in a class.

Overloaded constructors follow the same rules as discussed in the previous section for overloaded methods.

- *Overloaded constructors* must be defined using different argument lists.
- *Overloaded constructors* can’t be defined by just a change in the access levels.

Overloaded constructors

```
class Employee {
    String name;
    int age;
    Employee() { // no-argument constructor
        name = "John";
        age = 25;
    }
    Employee(String newName) { // constructor with one argument
        name = newName;
        age = 25;
    }
    Employee(int newAge, String newName) { // constructor with two arguments
        name = newName;
        age = newAge;
    }
}
```

Invoking an overloaded constructor using this

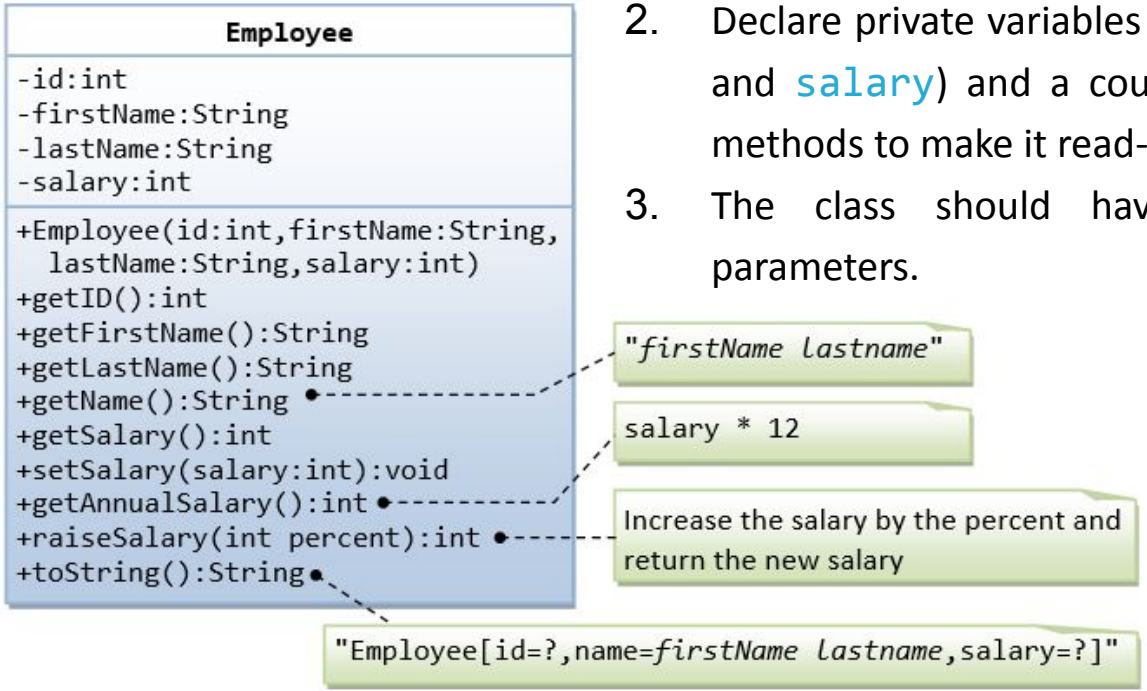
Overloaded constructors are invoked by using the keyword **`this`** - an implicit reference that's accessible to all objects that refer to an object itself.

```
class Employee {
    String name;
    int age;

    Employee() { // no-argument constructor
        //Invokes constructor that accepts two method arguments
        this("Mike", 0);
    }

    Employee(String newName, int newAge) { // constructor with two arguments
        name = newName;
        age = newAge;
    }
}
```

Rules to remember about constructor

- Overloaded constructors must be defined using different argument lists.
- Overloaded constructors can’t be defined by just a change in the access levels.
- Overloaded constructors may be defined using different access levels.
- A constructor can call another overloaded constructor by using the keyword this.
- A constructor can’t invoke a constructor by using its class’s name.
- If present, the call to another constructor must be constructor.
- You can’t call multiple constructors from a constructor.
- A constructor can’t be invoked from a method.

Exercise #4 The Employee Class



1. Create the class `Employee`.
2. Declare private variables (`id`, `firstName`, `lastName` and `salary`) and a couple of public getter and setter methods to make it read-write.
3. The class should have one constructor with all parameters.

Thank you for your attention!