4. Methods

Scope of Variables

1. The scope of a variable in Java refers to the part of the program where the variable is accessible. There are different scopes for variables in Java:

    1. Instance Variables - These are declared within a class but outside any method. They are accessible to all methods in the class and are initialised when an object of the class is created.
    2. Local Variables - These are declared within a method and are accessible only within that method. They are destroyed once the method completes execution.
    3. Static Variables - These are declared with the static keyword and are shared among all instances of a class. They are initialised when the class is loaded and are accessible to all methods in the class.

Local Variables

1. Local variables are defined within a method.
2. They may or may not be defined within code constructs such as if-else constructs, looping constructs, or switch statements.
3. Typically, you'd use local variables to store the intermediate results of a calculation.
4. Compared to the other three variable scopes listed previously, they have the shortest scope (life span).
5. TIP: The local variables topic is a favourite exam authors. You're likely to be asked a question that seems to be about a rather complex topic, such as inheritance or exception handling, but instead it'll be testing your knowledge on the scope of a local variable

VariableExample.java

```
public class VariableExample {

    // Class variable (static variable)
    static int classVariable = 10;

    // Instance variable
    int instanceVariable = 20;

    public static void main(String[] args) {
        // Local variable
        int localVariable = 30;

        // Accessing and modifying class variable
        System.out.println("Class variable before modification: " + classVariable);
        classVariable = 50;
        System.out.println("Class variable after modification: " + classVariable);
```

Local Variables

```
public void forwardReference(){
   int a = b;
   int b = 20;
}
```

```
public void forwardReference() {
   int b = 20;
   int a = b;
}
```

One of these won't compile. Which one?

1. The scope of a local variable depends on the location of its declaration within a method.
2. 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.
3. Local variables defined outside any of these constructs are accessible across the complete method.
4. The next section discusses the scope of method parameters.

Method Parameters

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

```
class Phone{
    private boolean tested ;
    public void setTested (boolean val ) {
        tested = val; // val is your method parameter
    }
    public boolean isTested ( )
        {return tested;
    }
}
```

Instance Variables

1. Instance is another name for an object. Hence, an instance variable is available for the life of an object.
2. An instance variable is declared within a class, outside all the methods.
3. It’s accessible to all the instance (or nonstatic) methods defined in a class.

Instance Variables

1. In the following example, the variable tested is an instance variable: it's defined within the class `Phone`, outside all the methods.
2. It can be accessed by all the methods of class `Phone`:

```
class Phone{
    private boolean tested ;
    public void set Tested (boolean val ) {
        tested = val; // tested is your method parameter
    }
    public boolean isTested ( )
        {return tested;
    }
}
```

Class variables(Static Variable)

1. A class variable is defined by using the keyword static.
2. A class variable belongs to a class, not to individual objects of the class.
3. A class variable is shared across all objects - objects don’t have a separate copy of the class variables.
4. You don’t even need an object to access a class variable.
5. It can be accessed by using the name of the class in which it’s defined

Object's Life Cycle

1. Creation - An object is created using the new keyword, which allocates memory for the object and initialises its fields.
2. Initialisation - The object's fields are initialised either with default values (if not explicitly initialised) or with the values provided in the constructor.
3. Utilisation - The object is used by calling its methods and accessing its fields.
4. Garbage Collection - When the object is no longer referenced by any part of the program, it becomes eligible for garbage collection, where the memory occupied by the object is reclaimed.

VARIABLES WITH THE SAME NAME IN DIFFERENT SCOPES

1. The fact that the scopes of variables overlap results in interesting combinations of variables within different scopes but with the same names.

2. Some rules are necessary to prevent conflicts. In particular, you can’t define a static variable and an instance variable with the same name in a class:

```
class MyPhone{

static boolean softKeyboard = true; // Won’t compile.

boolean softKeyboard = true; // Class variable and instance variable can’t be defined using the same name in a class.

}
```

VARIABLES WITH THE SAME NAME IN DIFFERENT SCOPES

1. Similarly, local variables and method parameters can't be defined with the same name.
2. The following code defines a method parameter and a local variable with the same name, so it won't compile:

```
// Won't compile.
// Method parameter and local variable can't be defined using the same name in a method.
void myMethod(int weight){
   int weight = 10;
}
```

VARIABLES WITH THE SAME NAME IN DIFFERENT SCOPES

1. What happens when you assign a value to a local variable that has the same name as an instance variable?
2. Does the instance variable reflect this modified value?
3. It should help you remember what happens when you assign a value to a local variable when an instance variable already exists with the same name in the class

Methods with Arguments and Return Values

1. Methods in Java can have parameters (arguments) and can return values.
2. Parameters are used to pass data to a method, while the return value is the result of the method's computation.

```
public int add(int a, int b) {
    return a + b;
}
```

Return type of a method

1. The return type of a method states the type of value that a method will return.
2. A method may or may not return a value.
3. One that doesn’t return a value has a return type of void.
4. A method can return a primitive value or an object of any class.
5. The name of the return type can be any of the eight primitive types defined in Java, a class, or an interface.

MethodExample.java

```
public class MethodExample {

    // Method with no return type and no parameters
    public static void greet() {
        System.out.println("Hello, world!");
    }

    // Method with return type and parameters
    public static int add(int a, int b) {
        return a + b;
    }

    // Method with return type and no parameters
    public static double getPi() {
        return 3.14159;
    }
```

MethodExample.java

1. In this example, the **`MethodExample`** class contains four methods:

1. greet: A method with no return type (void) and no parameters. It simply prints "Hello, world!" to the console.
2. add: A method with a return type of int and two int parameters. It returns the sum of the two parameters.
3. getPi: A method with a return type of double and no parameters. It returns the value of Pi (3.14159).
4. printDetails: A method with no return type (void) and two parameters (String name and int age). It prints the name and age to the console.

Method Parameters

1. 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.
2. You can pass multiple values to a method as input.
3. 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.

Return statement

1. The return statement in Java is used to exit a method and optionally return a value to the caller of the method.
2. When a return statement is encountered, the method stops executing and control is returned to the caller.
3. If the method has a return type other than void, the return statement must include a value of the appropriate type.

ReturnExample.java

```
public class ReturnExample {

    // Method with return type and parameters
    public static int add(int a, int b) {
        int sum = a + b;
        return sum;
    }

    // Method with return type and no parameters
    public static double getPi() {
        return 3.14159;
    }

    // Method with no return type and parameters
    public static void printDetails(String name, int age) {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        // No return statement needed in void methods, but you can use 'return;' to exit early
    }
}
```

ReturnExample.java

1. In this example, the ReturnExample class contains three methods:
    1. The **`add`** method takes two integers as parameters, calculates their sum, and returns the result using the return statement.
    2. The **`getPi`** method returns the value of Pi as a double using the return statement.
    3. The **`printDetails`** method takes a name and an age as parameters, prints them to the console, and does not return a value.
        1. Since it has a return type of void, no return statement is required at the end of the method, but you can use return; to exit early if needed.

When defining a return statement

Some items to note when defining a return statement:

1. For a method that returns a value, the return statement must be followed immediately by a value.
2. For a method that doesn't return a value (return type is void), the return statement must not be followed by a return value.
3. If the compiler determines that a return statement isn't the last statement to execute in a method, the method will fail to compile.
4. Do you think we've covered all the rules for defining a method? Not yet!
5. Do you think you can define multiple methods in a class with the same name? You can, but you need to be aware of some additional rules

Overloaded Methods

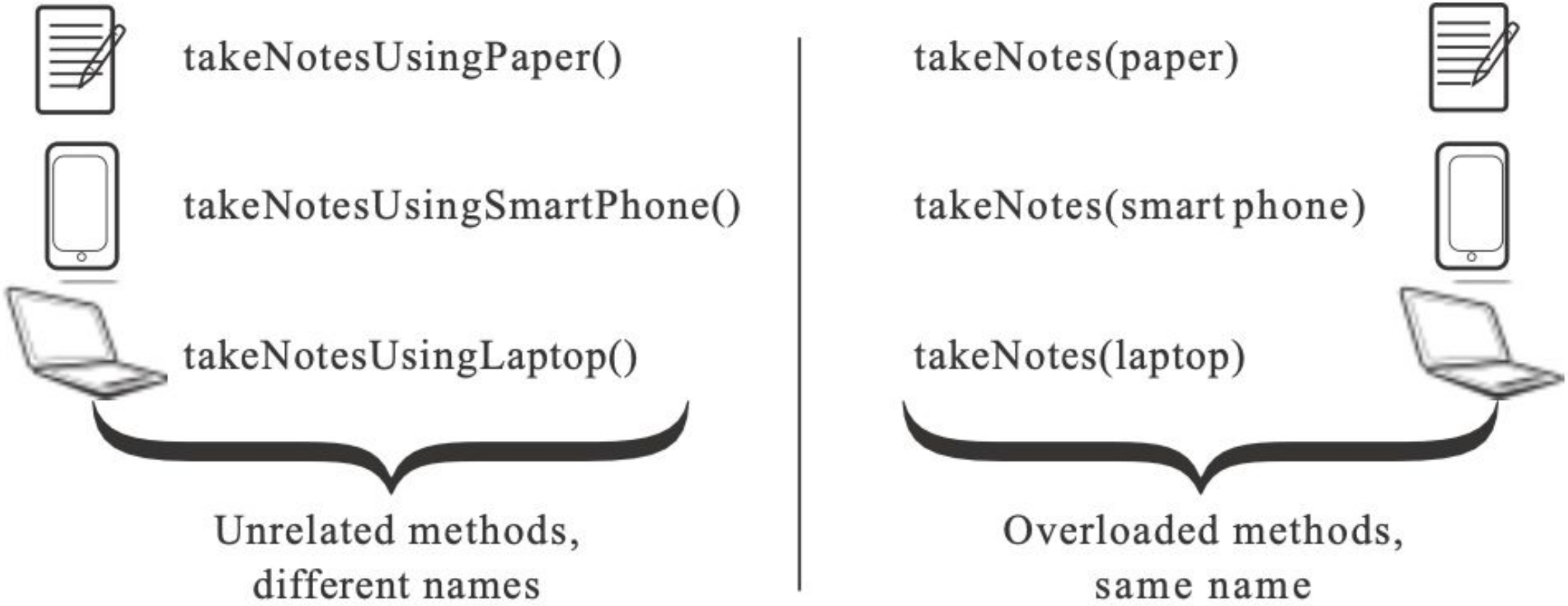
1. Overloaded methods are methods with the same name but different method parameter lists.
2. Imagine that you're delivering a lecture and need to instruct the audience to take notes using paper, a smartphone, or a laptop - whichever is available to them for the day.
3. One way to do this is give the audience a list of instructions as follows:

1. Take notes using paper.
2. Take notes using a smartphone.
3. Take notes using a laptop.

Overloaded Methods

1. Another method is to instruct them to “take notes” and then provide them with the paper, a smartphone, or a laptop they’re supposed to use.
2. Apart from the simplicity of the latter method, it also gives you the flexibility to add other media on which to take notes (such as one’s hand, some cloth, or the wall) without needing to remember the list of all the instructions.
3. This second approach, providing one set of instructions (with the same name) but a different set of input values can be compared to using overloaded methods in Java, as shown

Overloaded Methods



Overloaded Methods

Here are a few rules for defining overloaded methods:

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

Overloaded Methods

1. Method overloading is the ability to define multiple methods in a class with the same name but different parameters.
2. Java determines which method to call based on the number and types of arguments provided.

```
public int add(int a, int b) {
    return a + b;
}

public double add(double a, double b) {
    return a + b;
}
```

Constructors of a Class

1. Constructors are special methods in Java used for initialising objects.
2. They have the same name as the class and do not have a return type.

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }
}
```

Accessing Object Property

1. Object Properties (instance variables) can be accessed using the dot (.) operator followed by the field name.

```
MyClass obj = new MyClass(10);

int value = obj.value; // Accessing the value field of obj
```

Accessing Object Properties
Person.java

```
public class Person {
    // Private fields (variables) to store name and age
    private final String name;
    private final int age;

    // Constructor with parameters to initialize name and age
    public Person(String name, int age) {
        // Use 'this' keyword to refer to the current object's fields
        // Initialize the name and age fields with the values passed as parameters
        this.name = name;
        this.age = age;
    }
```

Passing by Value vs by Reference

Passing by Value: When you pass a primitive type (like int, double, char, etc.) to a method, you're passing a copy of the actual value. Any changes made to the parameter within the method won't affect the original value outside the method. This is because primitives hold their values directly, rather than referring to an object in memory

```
public static void main(String[] args) {
        int x = 10;
        increment(x);
        System.out.println(x); // Output: 10 (unchanged)
    }
    public static void increment(int num) {
        num = num + 1;
    }
```

Passing by Value vs by Reference

Passing by Reference(or rather, Passing Object Reference(Memory address) by Value): When you pass an object to a method, you're passing the reference (address) of the object, not the actual object itself. This reference still points to the same object in memory. If you modify the object's state within the method, those changes will be reflected outside the method because both the original reference and the copied reference point to the same object in memory.

```
public static void main(String[] args) {
        StringBuilder str = new StringBuilder("Hello");
        appendWorld(str);
        System.out.println(str); // Output: HelloWorld
    }
    public static void appendWorld(StringBuilder s) {
        s.append("World");
    }
```

OPTIONAL READ:
https://medium.com/javarevisited/is-java-pass-by-reference-or-pass-by-value-f03562367446

Passing Objects and Primitives to Methods

1. You can pass objects and primitive data types to methods.
2. When an object is passed, it is passed by reference, meaning changes to the object's state inside the method affect the original object.
3. Primitive data types are passed by value, meaning a copy of the value is passed to the method.

PassingObjects.java

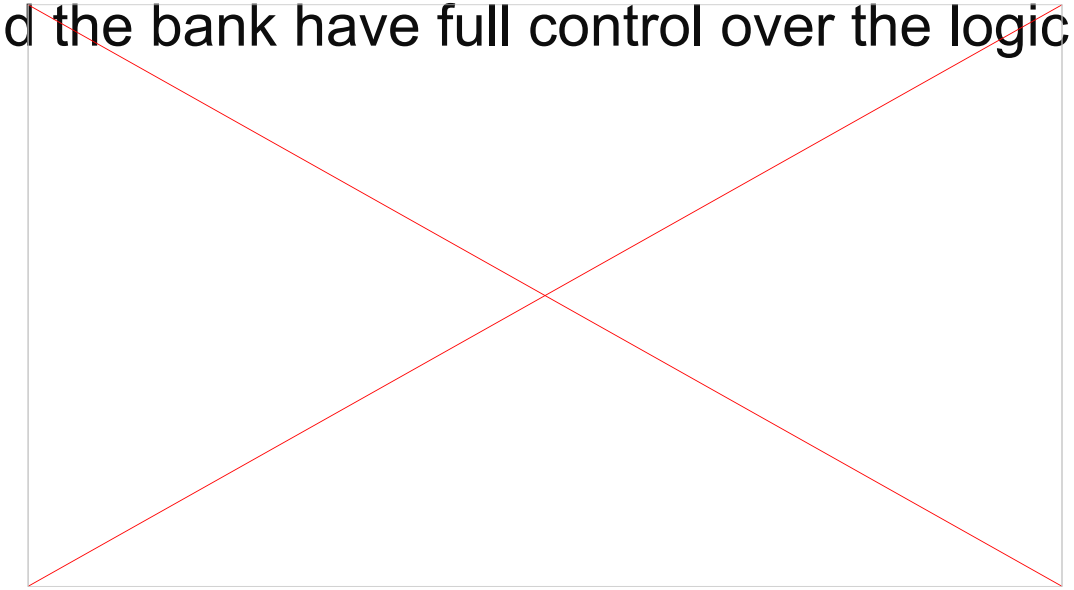
```
public class PassingObjects {
    // Private field (variable) to store the name
    private String name;

    // Constructor to initialize the name
    public PassingObjects(String name) {
        // Use 'this' keyword to refer to the current object's field
        // Initialize the name field with the value passed as a parameter
        this.name = name;
    }
```

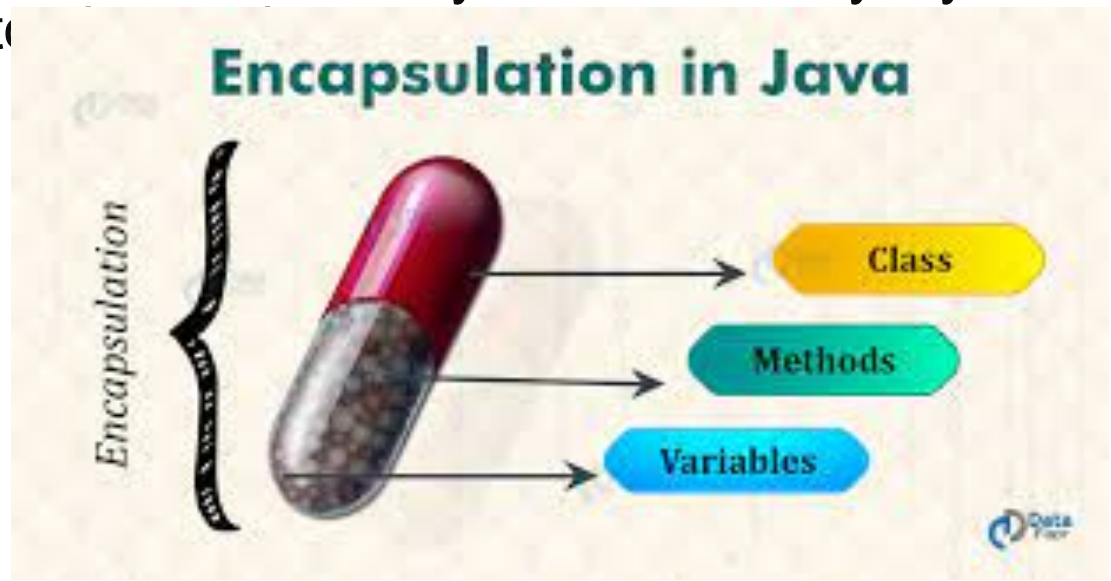
Read and write instance variables

1. The OCA Java SE 8 Programmer I exam will test you on how to read values from and write them to instance variables of an object, which can be accomplished by any of following:
    1. Using methods to read and write object fields
    2. Using constructors to write values to object fields
    3. Directly accessing instance variables to read and write object fields
2. Exam tip: Although object fields can be manipulated by direct access, it isn't a recommended practice. It makes an object vulnerable to invalid data. Such a class isn't well encapsulated.

Encapsulation

1. Encapsulation is one of the four pillars(principles) of Object Oriented Programming(OOP).
2. It aims to restrict direct access to some of the object's components and only allow access through the methods provided by the class, thus hiding the internal state of an object from outside interference.
3. For example let's suppose you downloaded CIMB bank app on your phone, if CIMB allowed you to change your bank balance from your phone without even having money, you can make yourself millionaire, that's why they don't allow you to change your balance yourself (they hide/encapsulate) the bank balance variable so you don't modify it yourself and the bank have full control over the logic t





Applying Encapsulation to a Class

1. Encapsulation involves hiding the internal state of an object and only exposing it through public methods (getters and setters).
2. Using public methods to change the data would allow you to enforce your logic, if you let the user of your application modify the data directly then the user can put whatever data they want
3. This helps in ensuring data integrity and security.

```
public class Person {

    private String name;


    public String getName() {

        return name;

    }


    public void setName(String name) {

        this.name = name;

    }

}
```

Key features of encapsulation

1. Data Hiding Encapsulation hides the internal state (data) of an object from the outside world. This prevents direct modification of the object's state and ensures that the state can only be changed in a controlled manner, typically through methods provided by the class.

2. Access Control - Encapsulation allows you to control the access to the data by specifying which methods can modify or view the data. This helps in enforcing data integrity and security.

3. Modularity - Encapsulation promotes modularity by grouping related data and methods into a single unit (class). This makes the code easier to understand, maintain, and reuse.

4. Information Hiding - Encapsulation hides the implementation details of how data is stored or manipulated. This allows you to change the internal implementation(logic) of a class without affecting its external system, as long as the public methods remain the same.

BankAccount.java

```
public class BankAccount {
    // Private fields for account number and balance
    private final String accountNumber;
    private double balance;

    // Constructor to initialize account number and balance
    public BankAccount(String accountNumber, double
initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }
```

BankAccount.java

1. The `BankAccount` class encapsulates the account number and balance.
2. The `deposit` and `withdraw` methods provide controlled access to modify the balance.
3. The main method demonstrates creating a `BankAccount` object, depositing and withdrawing funds, and checking the balance.
4. This demonstrates how encapsulation allows us to control access to the internal state of an object and maintain its integrity.

Overloaded Constructors

1. In the same way in which you can overload methods in a class, you can also overload the constructors in a class.
2. Overloaded constructors follow the same rules as discussed inthe previous section for overloaded methods. Here's a quick recap:
    1. Overloaded constructors must be defined using different argument lists.
    2. Overloaded constructors can't be defined by just a change in the access levels.

Overloaded Constructors: Rules to remember

1. Overloaded constructors must be defined using different argument lists.
2. Overloaded constructors can’t be defined by just a change in the access levels.
3. Overloaded constructors may be defined using different access levels.
4. A constructor can call another overloaded constructor by using the keyword this.
5. A constructor can’t invoke a constructor by using its class’s name.
6. If present, the call to another constructor must be the first statement in a constructor.
7. You can’t call multiple constructors from a constructor.
8. A constructor can’t be invoked from a method (except by instantiating a class using the new keyword)