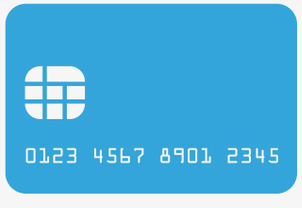
OBJECT ORIENTED PROGRAMMING IN DEPTH

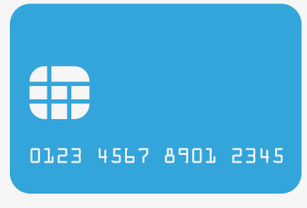
PRIMARY CONCEPTS: CLASS AND OBJECT

- Class describes template (blueprint) of something with state and behaviour
- Object is concrete instance of that class with set state

EXAMPLE: BANK CARD (STATE)

Class		Object
A. Bank Name	→	A. Citadele Banka
B. Payments Processor	→	B. Master Card
C. Name on Card	→	C. John Doe
D. Card Number	→	D. 5224 9989 7556 2871
E. Expiration Date	→	E. 12/2022
F. Security Code	→	F. 218

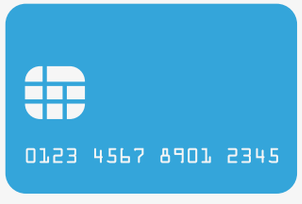




EXAMPLE: BANK CARD (BEHAVIOUR)

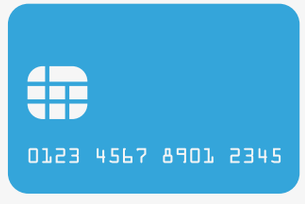
Class

A. Get balance
B. Deposit funds
C. Withdraw funds



Object

A. Get balance
B. Deposit funds
C. Withdraw funds



CLASS DECLARATION IN JAVA: SYNTAX

```
class ClassName {

    type variable1;
    type variable2;
    ...
    type variableN;

    method1() {}
    method2() {}
    ...
    methodN() {}

}
```

CLASS DECLARATION IN JAVA: EXAMPLE BREAKDOWN

Special keyword

Class name

Class variables (fields) defining state

Class methods defining behaviour

```
class Dog {

    String name;
    int age;
    boolean hungry;

    void bark() {

    }

    void sleep() {

    }

    void eat() {

    }

}
```

OBJECT INSTANTIATION IN JAVA: SYNTAX

- Object instantiation without assignment

```
new Class();
```

- Object instantiation with assignment

```
Class var = new Class();
```

OBJECT INSTANTIATION IN JAVA: SYNTAX

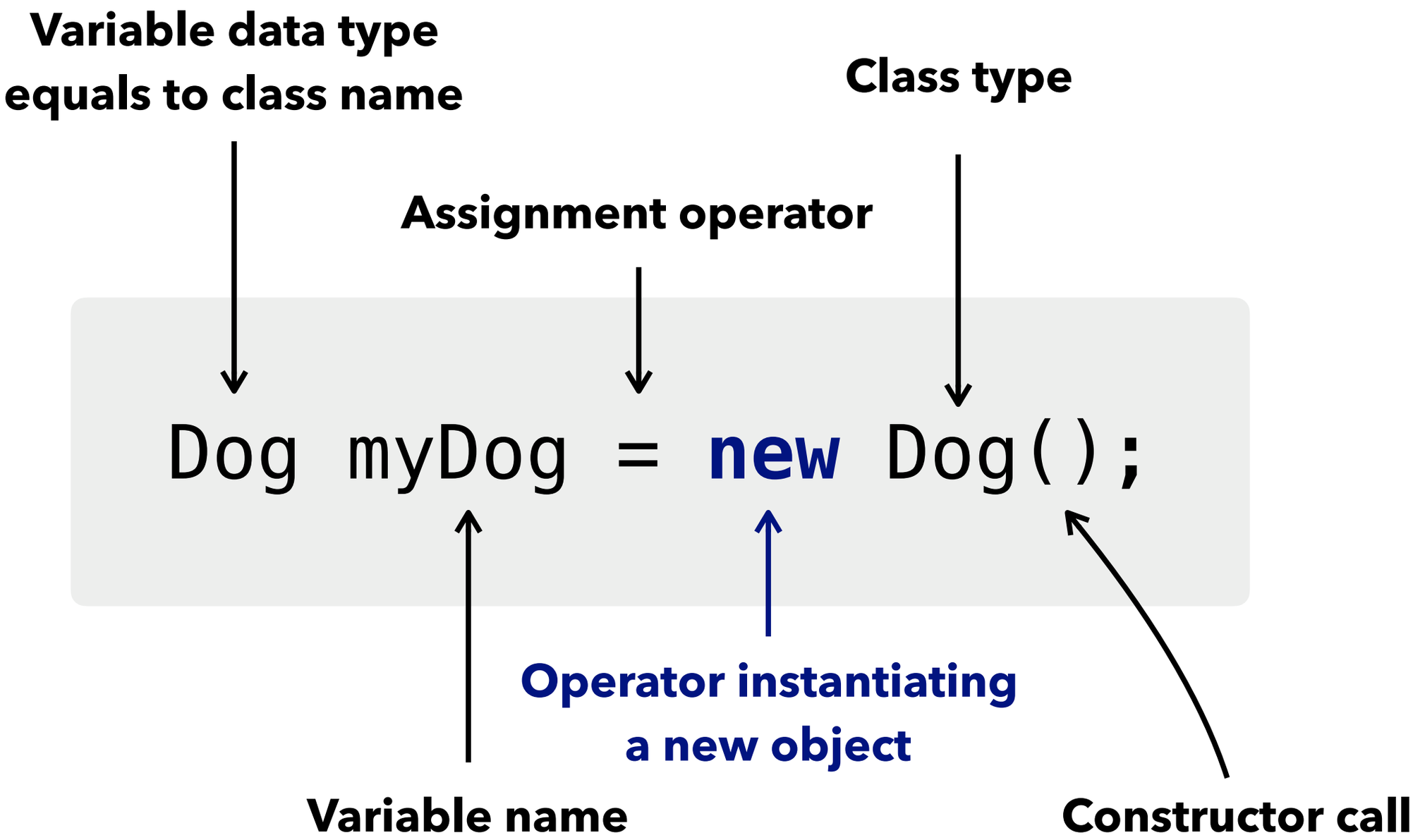
- Object instantiation without assignment

```
new Dog();
```

- Object instantiation with assignment

```
Dog myDog = new Dog();
```

OBJECT INSTANTIATION BREAKDOWN



THREE-STEP PROCESS OF OBJECT CREATION

1. Declaration - object variable declaration of a class type
2. Instantiation - the process of creating an object with new operator
3. Initialisation - the process of object construction by setting its initial state

CONSTRUCTORS

- Every class has a constructor
- If explicit constructor(s) is not specified in code, Java Compiler will generate default constructor implicitly
- Each time a new object is created, at least one constructor will be invoked
- Each defined constructor must have unique signature (i.e. ordered number and type of arguments)

CONSTRUCTOR DECLARATION IN JAVA: EXAMPLE BREAKDOWN

Explicit default constructor without arguments →

```
public class Dog {

    private String name;

    public Dog() {

    }

    public Dog(String name) {
        this.name = name;
    }

}
```

← **Explicit constructor with argument and initialisation**

MEMORY
OVERVIEW

MEMORY TYPES

- Java Heap Memory
  - Created objects are stored in the heap space
  - Lives from the start till the end of application execution
  - Objects stored in heap are globally accessible
- Java Stack Memory
  - Contains local primitive variables and reference variables to objects in heap space
  - Lives only within method execution, short-lived
  - Bound to the current execution thread

METHODS OVERVIEW

METHOD DEFINITION

- Java method is a collection of statements that are grouped together to perform an operation
  - Invoking *System.out.println()* method actually executes several statements in order to display a message on the console
- Describes behaviour of class or actions that object can perform
- Method either produces output or not

METHOD DECLARATION IN JAVA: SYNTAX

Defines the access type of the method

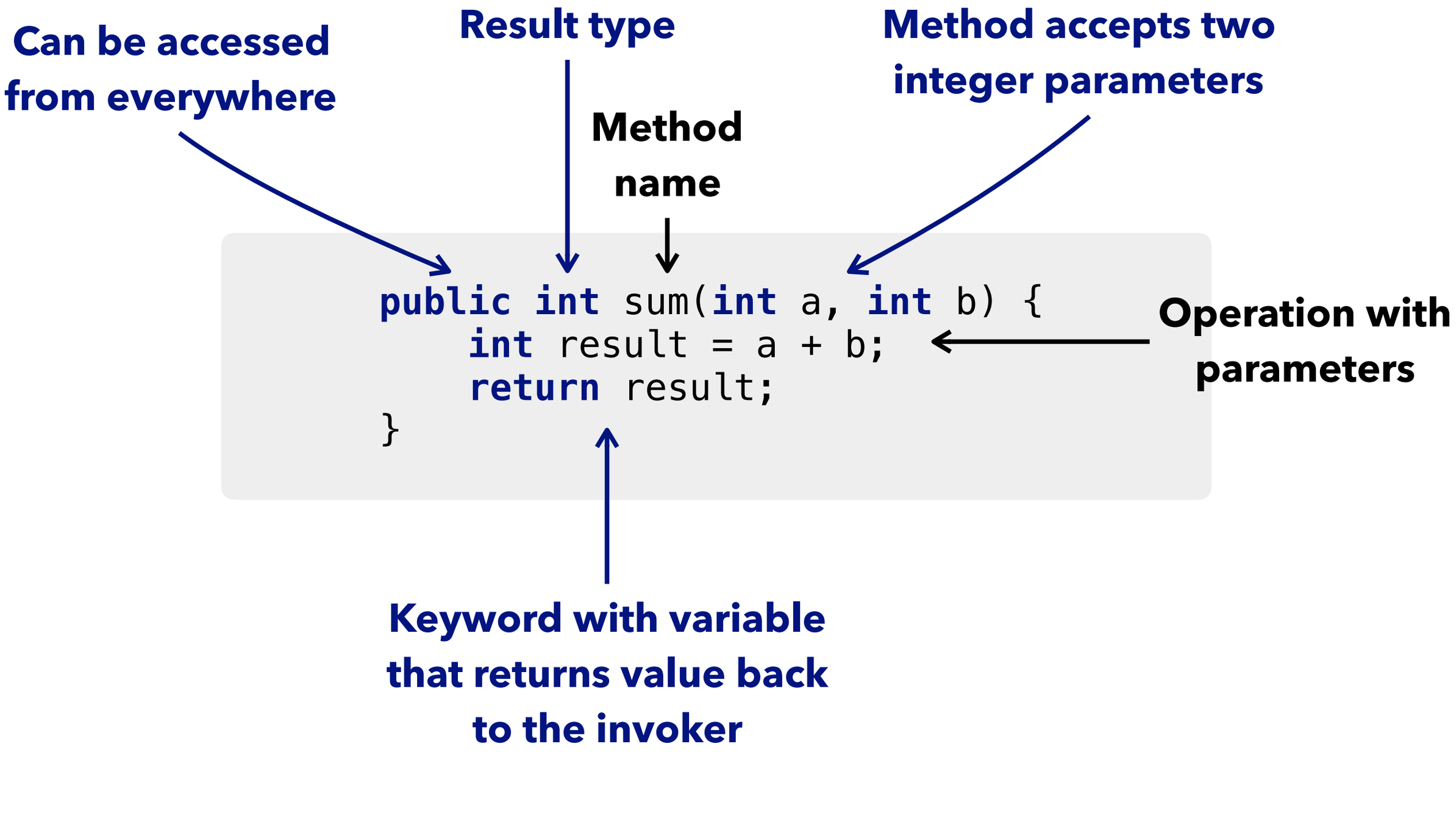
Defines method name

```
modifier returnType methodName (arg1, arg2, ..., argN) {
    //body
}
```

Specifies return type for methods producing output

List of parameters, it is type, order and number

METHOD DECLARATION IN JAVA: WITH RETURN EXAMPLE



METHOD DECLARATION IN JAVA: WITHOUT RETURN EXAMPLE

Method name

Keyword meaning no return value

Method accepts two integer parameters

```
public void printSum(int a, int b) {
    int result = a + b;
    System.out.println("Sum of numbers is" + result);
}
```

Operations with parameters

A BIT MORE ABOUT RETURNING RESULT

- After completion method returns to the code that invoked it
- Whether method returns value or not is declared in method signature
  - When type is void - return statement is unnecessary, however can be stated
  - Other type - return statement is necessary

ACCESSING AND CHANGING OBJECT STATE: GETTERS & SETTERS

- In OOP another party should not be able to access object state directly
- To keep things safe, one can
  - Retrieve object state via get methods (getters)
  - Change object state via set methods (setters)

GETTERS & SETTERS DECLARATION

```java
public class Person {

    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Getters

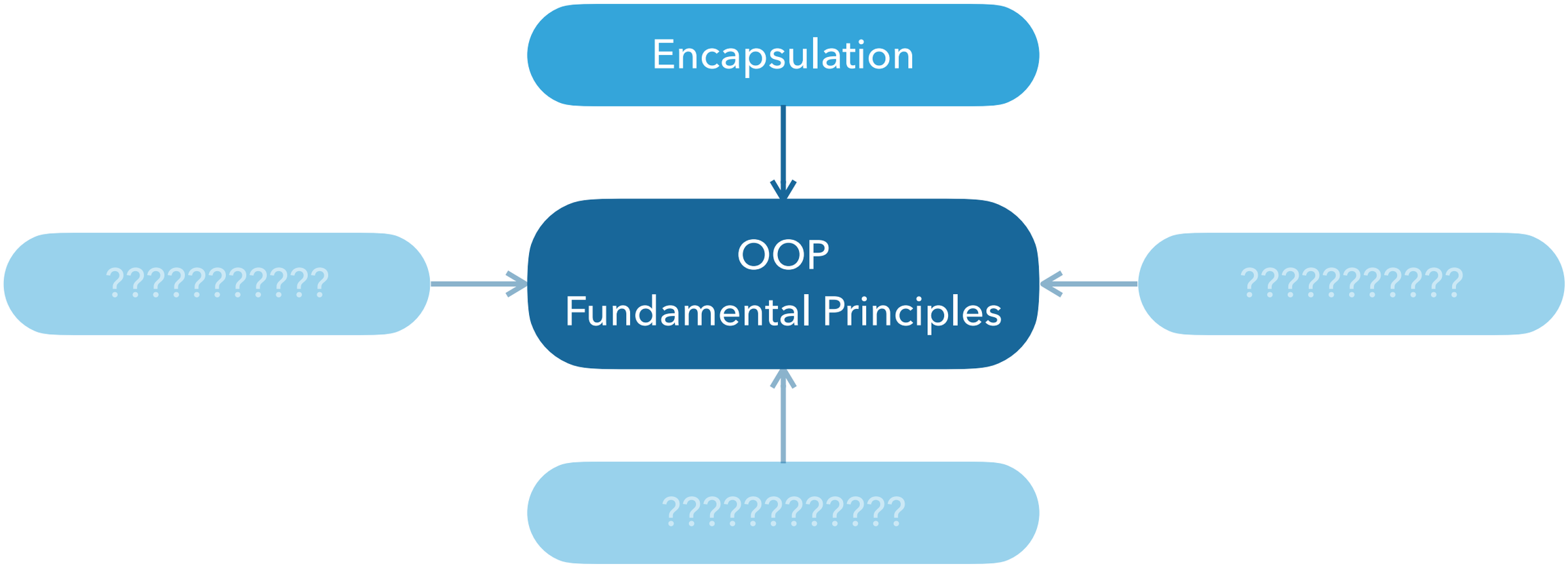
Setters

GETTERS & SETTERS USAGE

```
public class PersonTest {

    public static void main(String[] args) {

        Person person = new Person();
        person.setName("John Doe");
        person.setAge(32);

        String personName = person.getName();
        int personAge = person.getAge();

        System.out.println("His name is " + personName);
        System.out.println("He is " + personAge + " years old");

    }

}
```

CONCEPTS OF OBJECT ORIENTED PROGRAMMING

FOUR PILLARS OF OBJECT ORIENTED PROGRAMMING



ENCAPSULATION OVERVIEW

- Binding of data and behaviour together in a single unit
- Data is not accessed directly, but through the methods present inside class
- Makes the concept of data hiding possible

ACCESS MODIFIERS OVERVIEW

- Specifies which classes can access a given class and its fields, constructors and methods
- Classes, fields, constructors and methods can have one of four different access modifiers:
  - private
  - default (package private)
  - protected
  - public

PRIVATE ACCESS MODIFIER: SUMMARY

- When element is declared as private, then only code inside the same class can access it
- Declarable code elements:
  - Fields (variables)
  - Methods
  - Constructors
- Restricted code elements:
  - Classes

DEFAULT (PACKAGE PRIVATE) ACCESS MODIFIER: SUMMARY

- When element is declared as package private, then only code inside the same class or within the same package can access it
- Declarable code elements:
  - Fields (variables)
  - Methods
  - Constructors
  - Classes

PUBLIC ACCESS MODIFIER: SUMMARY

- When element is declared as public, then all code regardless of location can access it
- Declarable code elements:
  - Fields (variables)
  - Methods
  - Constructors
  - Classes

BASIC COUNTER: REQUIREMENTS

- State
  1. Current counter value cannot be accessed directly
- Behaviour
  2. Can increment, decrement and clear counter value
  3. Can set counter value to any specified positive number (otherwise set to 0)
  4. Can be constructed only within the same package

1. BASIC COUNTER: NO DIRECT ACCESS TO STATE

Hide internal state of counter by marking it as private

Allow external access by providing getter method

```
public class BasicCounter {

    private int counter;

    public int getCounter() {
        return counter;
    }
}
```

2. BASIC COUNTER: PRIMARY BEHAVIOUR

Control counter from outside without direct access to its state

```
public class BasicCounter {

    ...

    public void increment() {
        counter++;
    }

    public void decrement() {
        counter--;
    }

    public void clear() {
        counter = 0;
    }

}
```

3. BASIC COUNTER: SECONDARY BEHAVIOUR

Only counter knows about validation rules

```
public class BasicCounter {

    ...

    public void setCounter(int counter) {
        if (isPositive(counter)) {
            this.counter = counter;
        } else {
            clear();
        }
    }

    private boolean isPositive(int value) {
        return value > 0;
    }

}
```

4. BASIC COUNTER: CONSTRUCTION LIMITATIONS

No access modifier specified means it can be called within the same package

```
public class BasicCounter {
    ...
    BasicCounter() {
    }
    ...
}
```

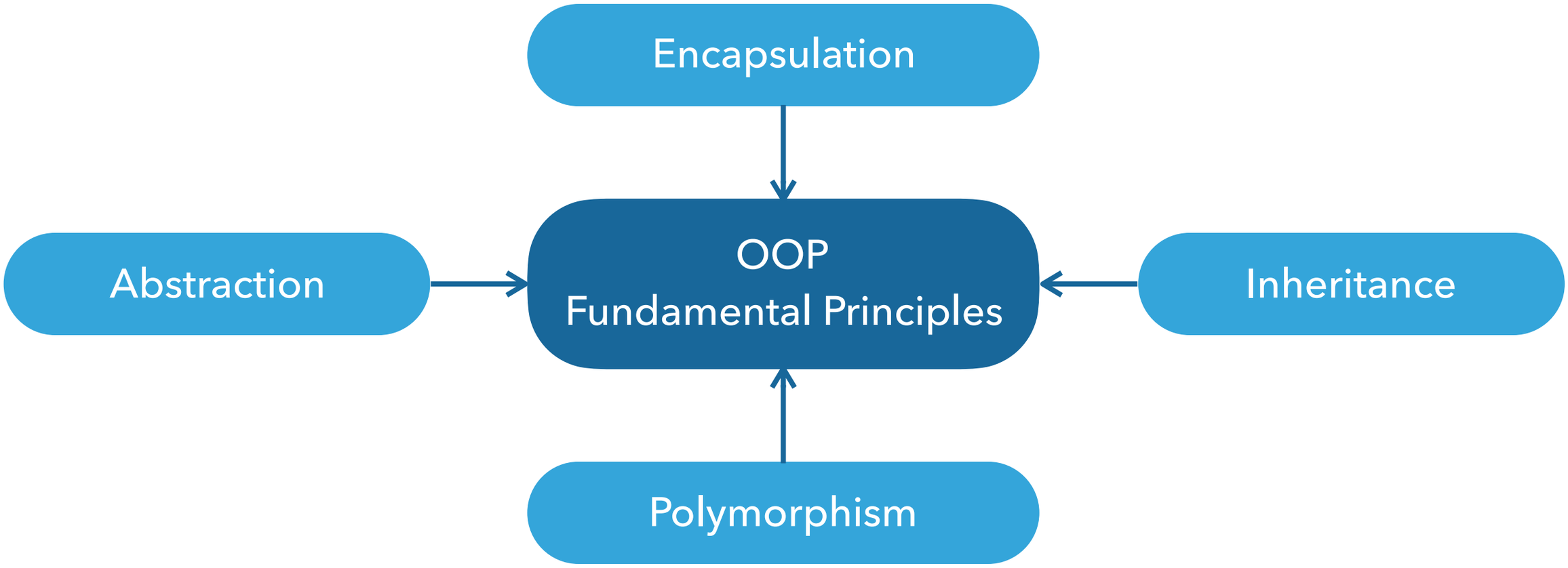
Empty constructor

BASIC COUNTER: FINAL RESULT

```java
public class BasicCounter {

    private int counter;

    BasicCounter() {
    }

    public int getCounter() {
        return counter;
    }

    public void setCounter(int counter) {
        if (isPositive(counter)) {
            this.counter = counter;
        } else {
            clear();
        }
    }

    public void increment() {
        counter++;
    }

    public void decrement() {
        counter--;
    }

    public void clear() {
        counter = 0;
    }

    private boolean isPositive(int value) {
        return value > 0;
    }

}
```

EXTENDED CONCEPTS OF OBJECT ORIENTED PROGRAMMING

FOUR PILLARS OF OBJECT ORIENTED PROGRAMMING



INHERITANCE

OVERVIEW

INHERITANCE OVERVIEW

- The process by which one class acquires the properties (data members or fields) and behaviour (methods) of another class is called inheritance
- The aim is to provide the reusability of code so that a class has to write only unique features

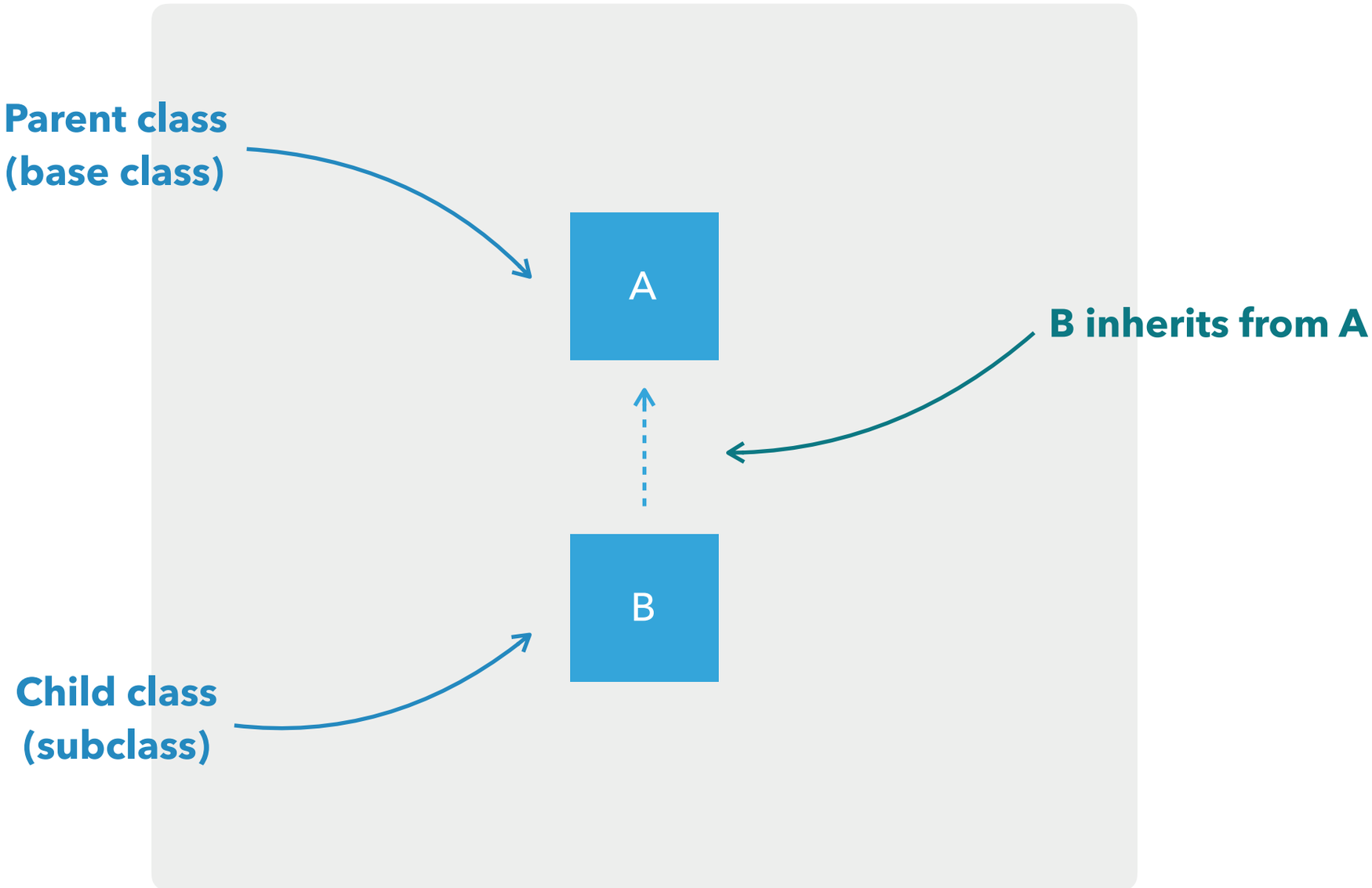
INHERITANCE CONCEPTS

- Child class
  - The class that extends the features of another class is known as child class, subclass or derived class
- Parent class
  - The class whose properties and functionalities are inherited by another class is known as parent class, superclass or base class

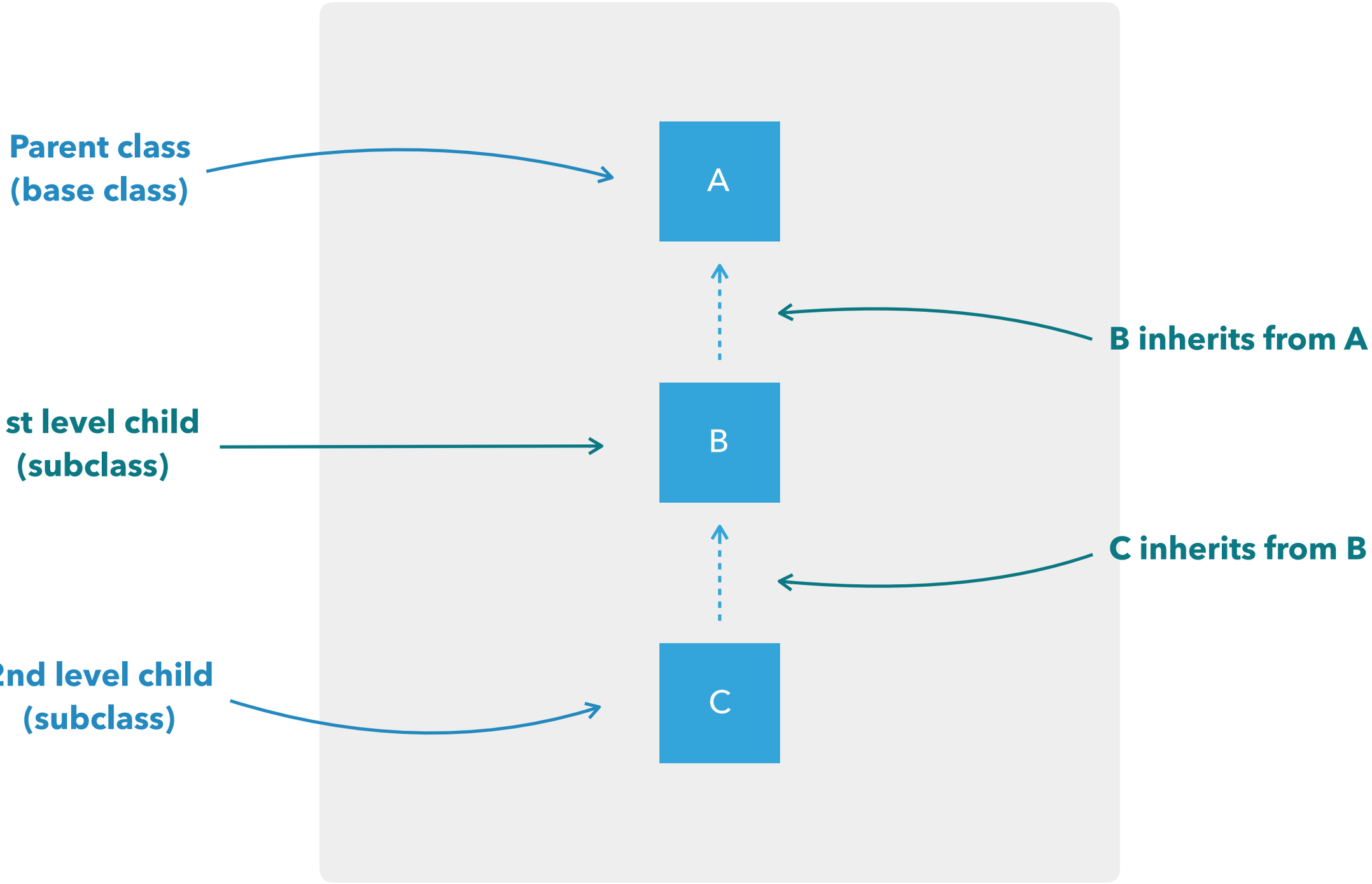
JAVA TYPES OF INHERITANCE: SUMMARY

- Single inheritance
  - Refers to a child and parent class relationship where a class extends the another class
- Multilevel inheritance
  - Refers to a child and parent class relationship where a class extends the child class
- Hierarchical inheritance
  - Refers to a child and parent class relationship where more than one classes extends the same class
- Hybrid inheritance
  - Combination of more than one types of inheritance in a single program

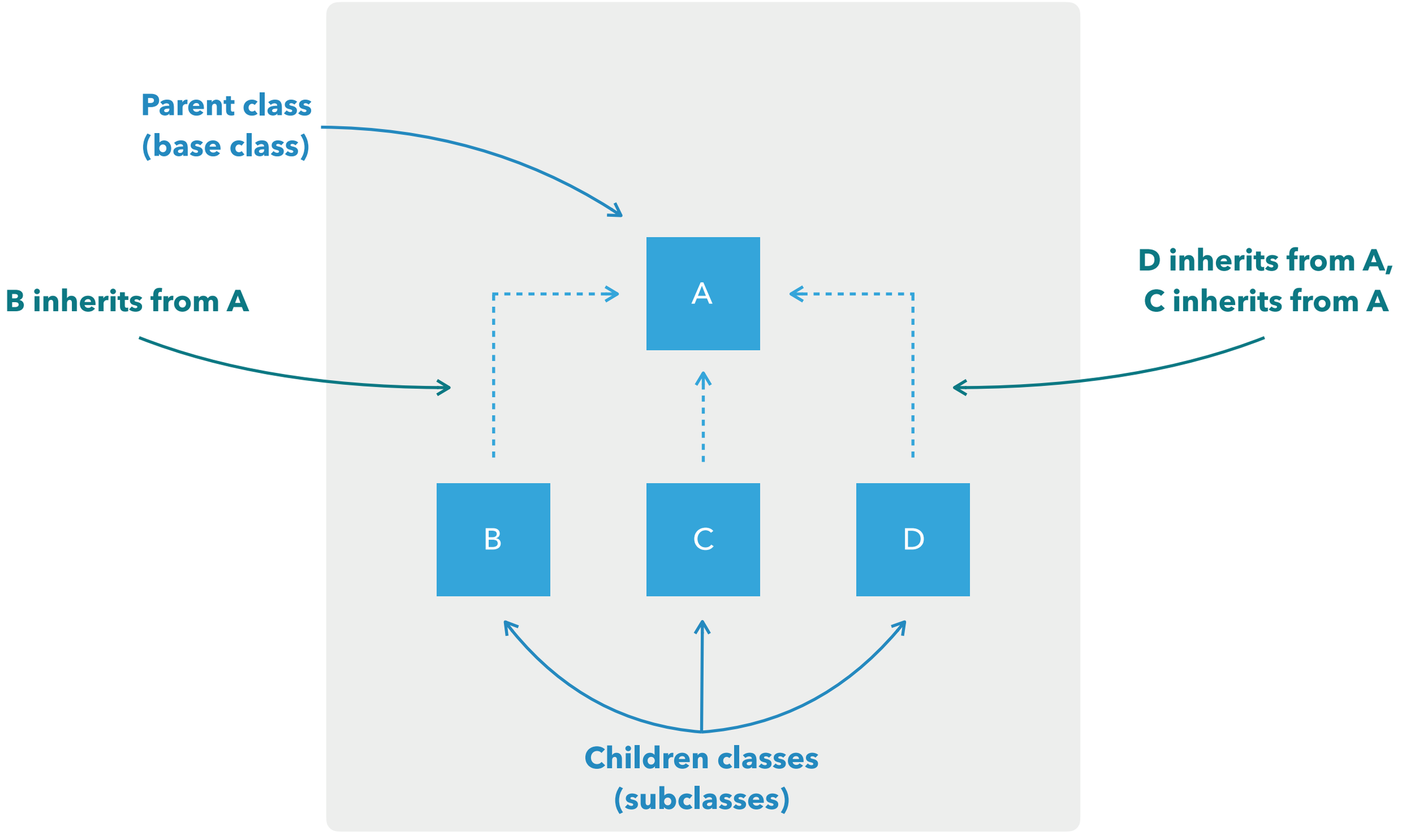
JAVA TYPES OF INHERITANCE: SINGLE INHERITANCE



JAVA TYPES OF INHERITANCE: MULTILEVEL INHERITANCE



JAVA TYPES OF INHERITANCE: HIERARCHICAL INHERITANCE



1. INHERITANCE: CODE EXAMPLE

Protected allows subclasses access fields or methods

```java
public class Bicycle {

    protected String brand;
    protected int speed;

    public Bicycle(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    public void accelerate() {
        this.speed++;
    }

    public void decelerate() {
        this.speed--;
    }

    @Override
    public String toString() {
        return "Bicycle{" +
                "brand='" + brand + '\'' +
                ", speed=" + speed +
                '}';
    }

}
```

2. INHERITANCE: CODE EXAMPLE

Subclass

Keyword stating inheritance process

Base class

Call parent's constructor

```
public class MountainBicycle extends Bicycle {

    protected int gear;

    public MountainBicycle(String brand, int speed, int gear) {
        super(brand, speed);
        this.gear = gear;
    }

    public void changeGear(int gear) {
        this.gear = gear;
    }

    @Override
    public String toString() {
        return "MountainBicycle{" +
                "gear=" + gear +
                ", brand='" + brand + '\'' +
                ", speed=" + speed +
                '}';
    }

}
```

3. INHERITANCE: CODE EXAMPLE

Code

```
Bicycle bicycle = new Bicycle("Pinarello", 15);
MountainBicycle mountainBicycle = new MountainBicycle("BMC", 42, 2);

System.out.println(bicycle);
System.out.println(mountainBicycle);
```

Console output

```
Bicycle{brand='Pinarello', speed=15}
MountainBicycle{gear=2, brand='BMC', speed=42}
```

4. INHERITANCE: CODE EXAMPLE

Code

```
System.out.println("Pedal to the metal!");
mountainBicycle.accelerate();

System.out.println(bicycle);
System.out.println(mountainBicycle);
```

Console output

```
Pedal to the metal!
Bicycle{brand='Pinarello', speed=15}
MountainBicycle{gear=2, brand='BMC', speed=43}
```

1. JAVA INHERITANCE: RULES AND LIMITATIONS

- Every class has default implicit Object superclass
  - In the absence of any other explicit superclass, every class is implicitly a subclass of Object class
    - Object class has no superclass
- Single inheritance principle
  - A superclass can has any number of subclasses, but a subclass can have only one superclass
  - Multiple inheritance with interfaces is permitted, even though java does not support multiple inheritance with classes

2. JAVA INHERITANCE: RULES AND LIMITATIONS

- Constructors are not inherited
  - A subclass inherits all members (fields, methods, and nested classes) from its superclass
  - Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass
- Private members inheritance
  - A subclass does not inherit the private members of its parent class
  - If superclass has public or protected methods (e.g. getters and setters) for accessing its private fields, these can also be used by subclass

JAVA INHERITANCE: RECAP

- In subclasses we can inherit members as is, modify them, hide them, or supplement them with new members:
  - Use inherited fields directly, just like any other fields
  - Declare new fields in the subclass that are not in the superclass
  - Write a new method in the subclass that has the same signature as the one in the superclass, thus overriding it (e.g. equals(), toString())
  - Declare new methods in the subclass that are not in the superclass
  - Write a subclass constructor that invokes the superclass constructor, either implicitly or by using the keyword super

ABSTRACTION

OVERVIEW

ABSTRACTION OVERVIEW

- The process where you show only relevant data and hide unnecessary details of an object from user
- Allows you to abstract from usage and rather outline generic object functionality
- Defines what object does instead of how

JAVA ABSTRACTION: SUMMARY

- Abstraction is achieved by two mechanisms:
  - Interfaces
    - Allows to achieve complete abstraction
  - Abstract classes
    - Allows to achieve partial abstraction

JAVA ABSTRACTION: INTERFACES OVERVIEW

- A bit like class, except:
  - Interface can only contain method signatures and fields
- Methods defined in interfaces cannot contain the implementation of method, only signature (return type, name, parameters, exceptions)
- Describes an object by actions it can perform
  - Sometimes interface names end with '-able' postfix (e.g. comparable)

1. JAVA ABSTRACTION: INTERFACE CODE EXAMPLE

Interface keyword instead of class

Interface name

```
public interface Singer {

    void sing();

}
```

Singers can sing, but we don't care how they do so

2. JAVA ABSTRACTION: INTERFACE CODE EXAMPLE

```
public class ElvisPresley implements Singer {

    ...

    @Override
    public void sing() {
        System.out.println("Love me tender, baby..");
    }

    ...

}
```

Special keyword to guarantee that we support interface specified behaviour

Concrete implementation of singers behaviour

3. JAVA ABSTRACTION: INTERFACE CODE EXAMPLE

```
public class MichaelJackson implements Singer {

    ...

    @Override
    public void sing() {
        System.out.println("Billie Jean is not my lover");
    }

    ...

}
```

4. JAVA ABSTRACTION: INTERFACE CODE EXAMPLE

```
public class BritneySpears implements Singer {

    ...

    @Override
    public void sing() {
        System.out.println("Hit me baby one more time");
    }

    ...

}
```

JAVA ABSTRACTION: ABSTRACT CLASS OVERVIEW

- Mostly like a class, except:
  - Can contain method signatures without implementation among other methods
  - Cannot be instantiated

1. JAVA ABSTRACTION: ABSTRACT CLASS CODE EXAMPLE

Marking class as abstract

```
public abstract class Shape {

    private String color;

    public Shape(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    abstract double area();

}
```

Subclasses must use constructor of parent class

Method signature that all children are forced to implement

2. JAVA ABSTRACTION: ABSTRACT CLASS CODE EXAMPLE

```
public class Circle extends Shape {

    private int radius;

    public Circle(String color, int radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    double area() {
        return 3.14 * radius * radius;
    }

}
```

Extending shape class with concrete details

Circle specific properties

Calling parent constructor with required params

Each concrete shape knows how to calculate its area

3. JAVA ABSTRACTION: ABSTRACT CLASS CODE EXAMPLE

```
public class Rectangle extends Shape {

    private int width;
    private int height;

    public Rectangle(String color, int width, int height) {
        super(color);
        this.width = width;
        this.height = height;
    }

    @Override
    double area() {
        return width * height;
    }

}
```

Rectangle specific properties

1. JAVA ABSTRACTION: INTERFACE VS ABSTRACT CLASS

- Type of methods
  - Interface can have only abstract methods (since Java 8 supports static and default methods as well)
  - Abstract class can have abstract and non-abstract methods
- Final variables
  - Variables declared in a Java interface are by default final
  - Abstract class may contain non-final variables

2. JAVA ABSTRACTION: INTERFACE VS ABSTRACT CLASS

- Type of variables
  - Interface has only static and final variables
  - Abstract class can have final, non-final, static and non-static variables
- Implementation
  - Interface can't provide the implementation of abstract class
  - Abstract class can provide the implementation of interface

3. JAVA ABSTRACTION: INTERFACE VS ABSTRACT CLASS

- Inheritance vs Abstraction
  - Interface can be implemented using keyword "implements"
  - Abstract class can be extended using keyword “extends”
- Multiple Implementation
  - Interface can extend another Java interface only
  - Abstract class can extend another Java class and implement multiple Java interfaces
- Accessibility of data members
  - Access modifiers of interface members are public by default and cannot be changed
  - Access modifiers of abstract class members can have any access modifiers (except private abstract methods)

POLYMORPHISM

OVERVIEW

POLYMORPHISM OVERVIEW

- Polymorphism is the ability of an object to take on many forms
- Capability of a method to do different things based on the object that it is acting upon
- Which implementation to be used is decided at runtime depending upon the situation

1. POLYMORPHISM: CODE EXAMPLE

Code

```
Singer elvis = new ElvisPresley();
Singer jackson = new MichaelJackson();
Singer spears = new BritneySpears();

elvis.sing(); jackson.sing(); spears.sing();
```

Console output

```
Love me tender, baby..
Billie Jean is not my lover
Hit me baby one more time
```

2. POLYMORPHISM: CODE EXAMPLE

Code

```
Singer[] singers = new Singer[2];
singers[0] = new ElvisPresley(); singers[1] = new BritneySpears();

for (Singer singer : singers) {
    singer.sing();
}
```

Console output

```
Love me tender, baby..
Hit me baby one more time
```

3. POLYMORPHISM: CODE EXAMPLE

Code

```
Shape circle = new Circle("Red", 3);
Shape rectangle = new Rectangle("Blue", 2, 4);

System.out.println("Circle area = " + circle.area());
System.out.println("Rectangle area = " + rectangle.area());
```

Console output

```
Circle area = 28.259999999999998
Rectangle area = 8.0
```