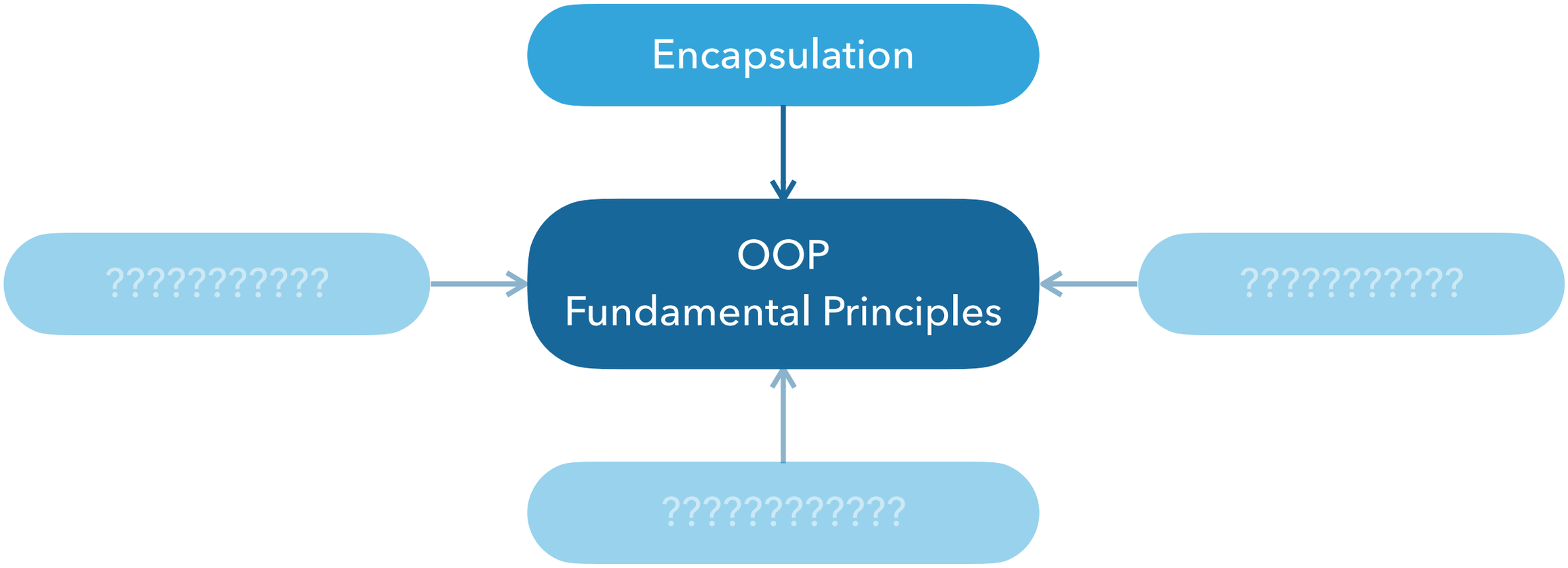
JAVAGURU INTRODUCTION TO JAVA

LESSON 7

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;
    }

}
```

OBJECT EQUALITY AND IDENTITY

OBJECT AND HEAP MEMORY REVISION

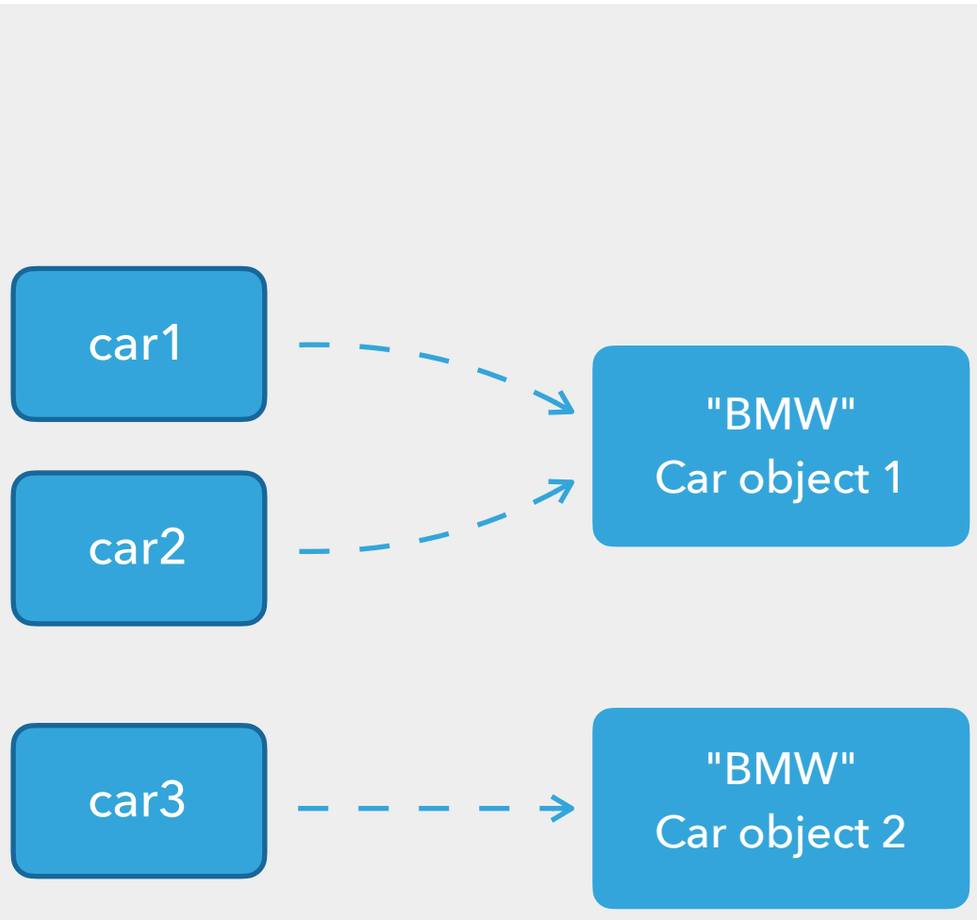
- When object is created, it is being stored in the heap memory
- To be able to locate an object, computer assigns it an address in the memory
- Every new object created gets a new address

REFERENCE AND OBJECTS IN HEAP MEMORY

Code view

```
Car car1 = new Car("BMW");
Car car2 = car1;
Car car3 = new Car("BMW");
```

Objects in memory view



REFERENCE EQUALITY: RELATIONAL OPERATOR

- Relational operator == used to compare two operands and determine whether the two operands are equal or not
- When used on referential type, we can see if both variables refer to the same object in the heap memory

REFERENCE EQUALITY: CODE EXAMPLE

```
Car car1 = new Car("BMW");
Car car2 = car1;
Car car3 = new Car("BMW");

if (car1 == car1) { //true
}

if (car1 == car2) { //true
}

if (car1 == car3) { //false
}
```

LOGICAL EQUALITY: METHOD EQUALS

- Every class by default has equals method that compares object method was called on with specified parameter
- Compares the data of the objects instead of the value of the references

LOGICAL EQUALITY: CODE EXAMPLE

```
Car car1 = new Car("BMW");
Car car2 = car1;
Car car3 = new Car("BMW");

if (car1.equals(car1)) { //true
}

if (car1.equals(car2)) { //true
}

if (car1.equals(car3)) { //false
}
```

SAME, BUT DIFFERENT, BUT STILL THE SAME

By default it compares by reference

Object class default equals method implementation

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

OVERRIDE DEFAULT BEHAVIOUR WITH CUSTOM LOGIC

- Default method implementation knows nothing about concrete class data, hence reference comparison by default
- Control what data of the class should be compared and how it should be done

OVERRIDE DEFAULT BEHAVIOUR: CODE EXAMPLE

```
public class Car {

    private String brand;

    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return Objects.equals(brand, car.brand);
    }

    ...

}
```

Check if both reference the same object

Check argument is not null and both are instances of the same class

Cast argument to the Car type

Specify which class fields to compare

LOGICAL EQUALITY AFTER EQUALS OVERRIDE: CODE EXAMPLE

```
Car car1 = new Car("BMW");
Car car2 = car1;
Car car3 = new Car("BMW");

if (car1.equals(car1)) { //true
}

if (car1.equals(car2)) { //true
}

if (car1.equals(car3)) { //true
}
```

TRICKY QUESTION: STRING INSTANTIATION

Instantiating String object without new keyword

```
String artist = "Justin Bieber";

String band = new String("Nickelback");
```

Instantiating String object with new keyword

EQUALITY DIFFERENCE

Reference equality

```
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");

if (s1 == s2) { //true
}

if (s1 == s3) { //false
}
```

Logical equality

```
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");

if (s1.equals(s2)) { //true
}

if (s1.equals(s3)) { //true
}
```

OBJECT TEXTUAL REPRESENTATION

WRITING OBJECT DETAILS IN THE CONSOLE: LONG WAY

Code

```
SmartPhone phone = new SmartPhone("Apple", "iPhone X");

System.out.println("Brand: " + phone.getBrand());
System.out.println("Model: " + phone.getModel());
```

Console output

```
Brand: Apple
Model: iPhone X

Process finished with exit code 0
```

WRITING OBJECT DETAILS IN THE CONSOLE: FAST WAY

Code

```
SmartPhone phone = new SmartPhone("Apple", "iPhone X");

System.out.println(phone);
```

Console output

```
lv.javaguru.lessons.l5.SmartPhone@1540e19d

Process finished with exit code 0
```

DEFAULT TO STRING METHOD BEHAVIOUR

Start with declared class name

Separate with @ symbol

```
public String toString() {
    return getClass().getName() + "@"
        + Integer.toHexString(hashCode());
}
```

End with HEX representation
of integer hash of the object

OVERRIDE DEFAULT BEHAVIOUR: CODE EXAMPLE

```
public class SmartPhone {

    private String brand;
    private String model;

    ...

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

}
```

WRITING OBJECT DETAILS AFTER OVERRIDE

Code

```
SmartPhone phone = new SmartPhone("Apple", "iPhone X");

System.out.println(phone);
```

Console output

```
SmartPhone{brand='Apple', model='iPhone X'}

Process finished with exit code 0
```

REFERENCES

- https://dzone.com/articles/object-identity-and-equality-in-java
- https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--
- https://users.soe.ucsc.edu/~eaugusti/archive/102-winter16/misc/howToOverrideEquals.html