**Reference** vs **Object** vs **Instance** vs **Class**

- A **class** is basically a blueprint for a house, using the blueprint (plans) we can build as many houses as we like based on those plans

- Each house you build (in other words **instantiate** using the **new** operator) is an object also known as an **instance**.

- Each house you build has an address (a physical location). In other words if you want to tell someone where you live, you give your address (perhaps written on a piece of paper). This is known as a **reference**.

- You can copy that **reference** as many times as you like but there is still just one house. In other words, we are copying the paper that has address on it not the house itself.

- We can pass **references** as parameters to **constructors** and **methods**

House blueHouse = new House(“blue”);

(variable) (instance)

- This creates a new **instance** of the House class.

- Remember House is a blueprint, and we are assigning it to the blueHouse **variable**. In other words, it is a **reference** to the **object** in memory.

House anotherHouse = blueHouse;

(reference) (object)

- This creates another **reference** to the same **object** in memory.

Method **Overloading**  vs **Overriding**

A) Overloading

- Method **overloading** means providing two or more separate methods in a class with the **same name** but **different parameters**

- Method return type may or may not be different and that allows us to **reuse** the same method name

- **Overloading** is very handy, it reduces duplicated code and we don’t have to remember multiple method names

- **Overloading** does not have anything to do with **polymorphism(**다형성**)** but Java developers often refer to overloading as Compile Time Polymorphism.

- In other words the compiler decided which method is going to be called based on the method name, return type and argument list

- We **can overload static** and **instance** methods (more about the difference between static and instance methods later)

- Usually **overloading** happens inside a single class, but a method can also be treated as **overloaded** in the subclass of that class

- That is because a **subclass inherits** one version of the method from the parent class and then the subclass can have another overloaded version of the method.

B) Overriding

- Method overriding means defining a method in a child class that already exists in the parent class with same signature (same name, same arguments)

- By extending the parent class, the child class gets all the methods defined in the parent class

- Method **overriding** is also known as **Runtime Polymorphism** and **Dynamic Method Dispatch**, because the method that is going to be called is decided at runtime by the JVM.

- When we **override** a method, it’s recommended to put **@Override** immediately above the method definition. This is an annotation that the compiler reads and will then show us an error if we don’t follow overriding rules correctly.

- We **can’t override static** methods, **only instance** methods.

Graphical user interface, diagram

Description automatically generated

Table

Description automatically generated

**Static** vs **Instance** Methods

A) Static Methods

- **Static methods** are declared using a **static** modifier

- **Static methods** can’t access instance methods and instance variables directly

- They are usually used for operations that don’t require any data from an instance of the class (from ‘this’)

- In **static methods** we can’t use the **this** keyword

- Whenever you see a method that **does not use instance variables** that method should be declared as a **static method**

- For example, main is a static method and it is called by the JVM when it starts an application

Text

Description automatically generated

B) Instance Methods

- **Instance methods** belong to an instance of a class

- To use an **instance method**, we have to instantiate the class first usually by using the **new** keyword

- **Instance methods** can access instance methods and instance variables directly

- **Instance methods** can also access static methods and static variables directly

Diagram

Description automatically generated

**Static Variables** vs **Instance Variables**

A) Static Variables

- Declared by using the keyword **static**

- **Static variables** are also known as **static member variables**

- Every instance of that class **shares** the same static variable

- If changes are made to that variable, all other instances will see the effect of the change

- **Static variables** are not used very often but can sometimes be very useful

- For example, when reading user input using **Scanner,** we will declare scanner as a static variable

- That way **static methods** can access it directly.

Graphical user interface, text

Description automatically generated

B) Instance Variables

- They **don’t** use the **static** keyword

- Instance variables are also known as fields or member variables

- **Instance variables** belong to an instance of a class

- Every instance has its own copy of an instance variable

- Every instance can have a different value (state)

- Instance variables represent the state of an instance

Text

Description automatically generated