



# Lesson 4

08.05.2023



# Naming Conventions

## Class Names

Class names should be nouns, as they represent “things” or “objects.” They should be mixed case (camel case) with only the first letter of each word capitalized, as in the following:

```
public class Fish {...}
```

## Interface Names

Interface names should be adjectives. They should end with “able” or “ible” whenever the interface provides a capability; otherwise, they should be nouns. Interface names follow the same capitalization convention as class names:

```
public interface Serializable {...}  
public interface SystemPanel {...}
```

## Method Names

Method names should contain a verb, as they are used to make an object take action. They should be mixed case, beginning with a lowercase letter, and the first letter of each subsequent word should be capitalized. Adjectives and nouns may be included in method names:

```
public void locate() {...} // verb  
public String getWayPoint() {...} // verb and noun
```

## Parameter and Local Variable Names

Parameter and local variable names should be descriptive lowercase single words, acronyms, or abbreviations. If multiple words are necessary, they should follow the same capitalization convention as method names:

```
public void printHotSpots(ArrayList spotList) {  
    int counter = 0;  
    for (String hotSpot : spotList) {  
        System.out.println("Hot Spot #"  
            + ++counter + ": " + hotSpot);  
    }  
}
```

## Constant Names

Constant names should be all uppercase letters, and multiple words should be separated by underscores:

```
public static final int MAX_DEPTH = 200;
```

## Enumeration Names


Enumeration names should follow the conventions of class names. The enumeration set of objects (choices) should be all uppercase letters:

```
enum Battery {CRITICAL, LOW, CHARGED, FULL}
```

## Acronyms

When using acronyms in names, only the first letter of the acronym should be uppercase and only when uppercase is appropriate:

```
public String getGpsVersion() {...}
```

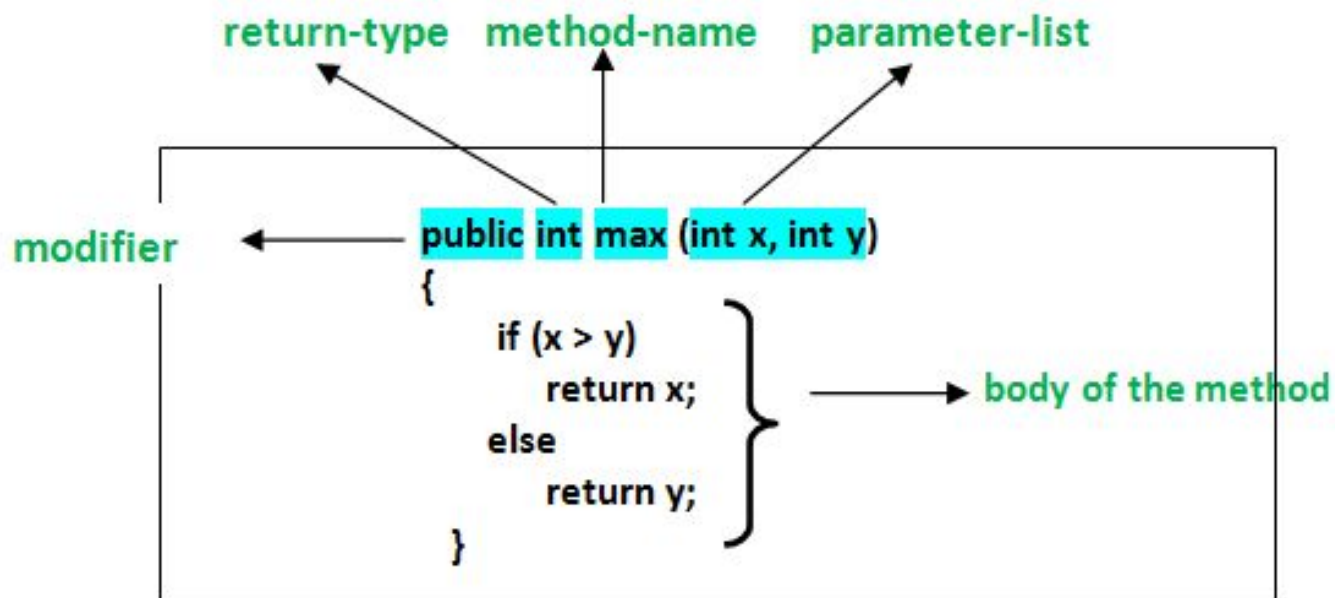


**Функція** – частина програми, яка має власне ім'я. Це ім'я можна використовувати у програмі як команду (така команда називається викликом функції). Під час виклику функції виконуються команди, з яких вона складається. Виклик функції може повертати значення (аналогічно операції) і тому може використовуватися у виразі поряд з операціями.

**Метод** - це функція, яка є частиною деякого класу, яка може виконувати операції над даними цього. У мові Java вся програма складається тільки з класів і функції можуть описуватись тільки всередині них. Саме тому всі функції мови Java є методами.

Функції використовуються у програмуванні, щоб зменшити його складність:

- 1.Замість того, щоб писати безперервну послідовність команд, якої незабаром перестав орієнтуватися, програму розбивають на підпрограми, кожна з яких вирішує невелике закінчене завдання, а потім велика програма складається з цих підпрограм (цей прийом називається декомпозицією).
- 2.Зменшується загальна кількість коду, тому що, як правило, одна функція використовується у програмі кілька разів.
- 3.Написана якимось і всебічно перевірена функція, можливо включена до бібліотеки функцій та використовуватись в інших програмах (при цьому не треба згадувати, як була запрограмована ця функція, достатньо знати, що вона робить).

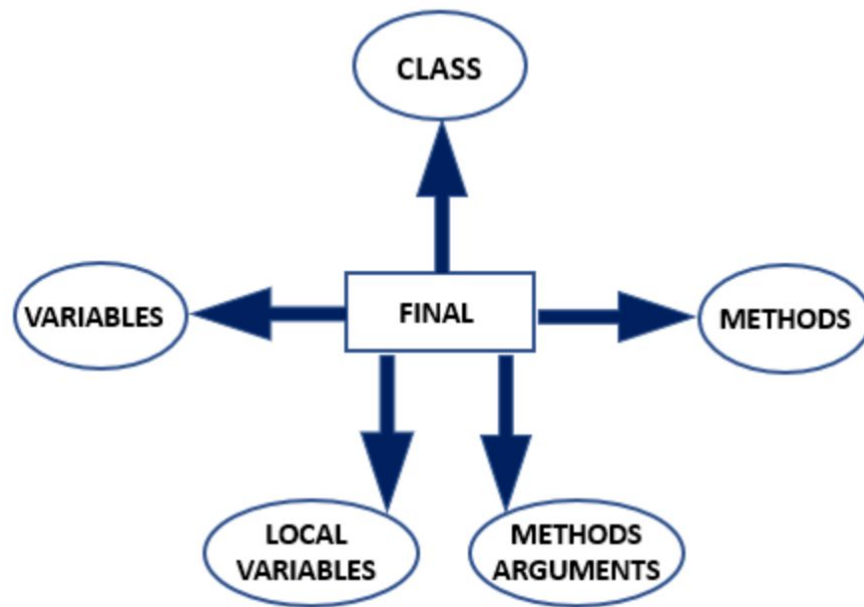




Access Modifiers	Non-Access Modifiers
<p>private default or No Modifier protected public</p>	<p>static final abstract synchronized transient volatile strictfp</p>



# 1. Final Non Access Modifiers



**Final** ключове слово використовується з класом, коли ми хочемо обмежити його успадкування будь-яким іншим класом. Наприклад, якщо у нас є **final** клас, то будь-яка спроба розширити цей клас може призвести до помилки під час компіляції.



## 2. Abstract Non-Access Modifier



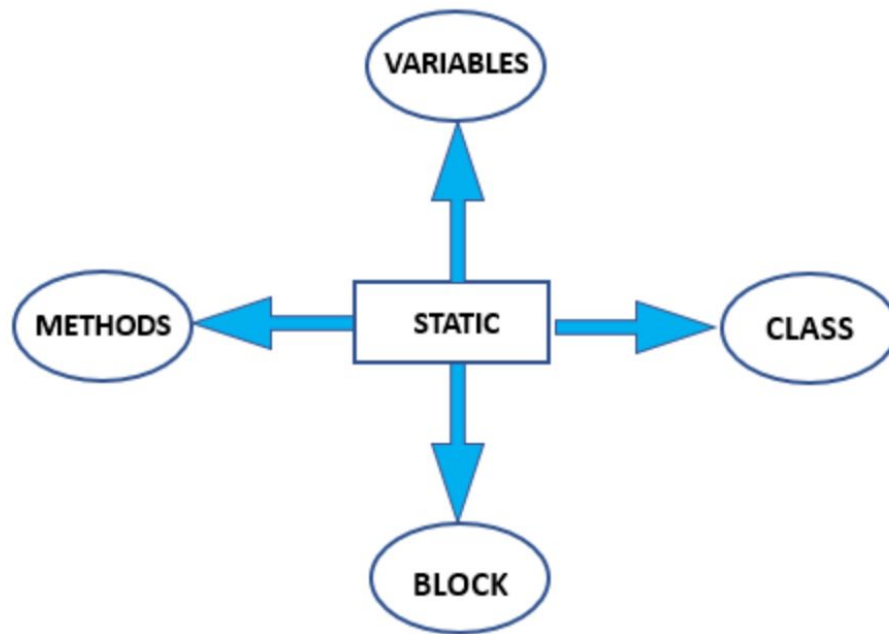
Клас оголошується як абстрактний, щоб вказати, що цей клас не може бути створений, що означає, що жодні об'єкти не можуть бути сформовані для цього класу, але можуть бути успадковані. Тим не менш, цей клас має конструктор, який буде викликатися всередині конструктора його підкласу. Він може містити як абстрактні, так і фінальні методи, де абстрактні методи будуть замінені в підкласі.

### 3. Synchronized Non-Access Modifier



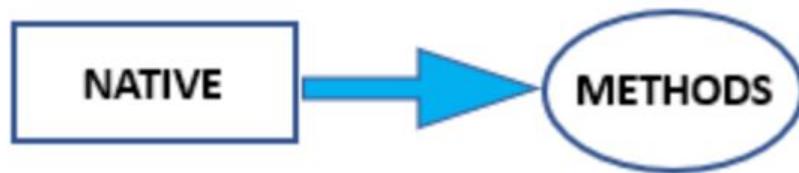
Це ключове слово допомагає запобігти одночасному доступу кількох потоків до одного методу, таким чином синхронізуючи потік програми та виводячи бажані результати за допомогою функції багатопоточності.

## 4. Static Non-Access Modifier



Ця змінна використовується для керування пам'яттю та для першого посилання під час завантаження класу. До цих членів ставляться на рівні класу; таким чином, їх не можна викликати за допомогою об'єкта; натомість ім'я класу використовується для посилання на них.

## 5. Native Non Access Modifier



Ключове слово `native` використовується лише з методами, щоб вказати, що конкретний метод написаний залежно від платформи. Вони використовуються для покращення продуктивності системи, а існуючий застарілий код можна легко використовувати повторно.

## 6. Strictfp Non-Access Modifier



Це ключове слово використовується для того, щоб результати операції над числами з плаваючою комою виводили однакові результати на кожній платформі. Це ключове слово не можна використовувати з абстрактними методами, змінними або конструкторами, оскільки вони не повинні містити операції.

## 7. Transient Non-Access Modifier

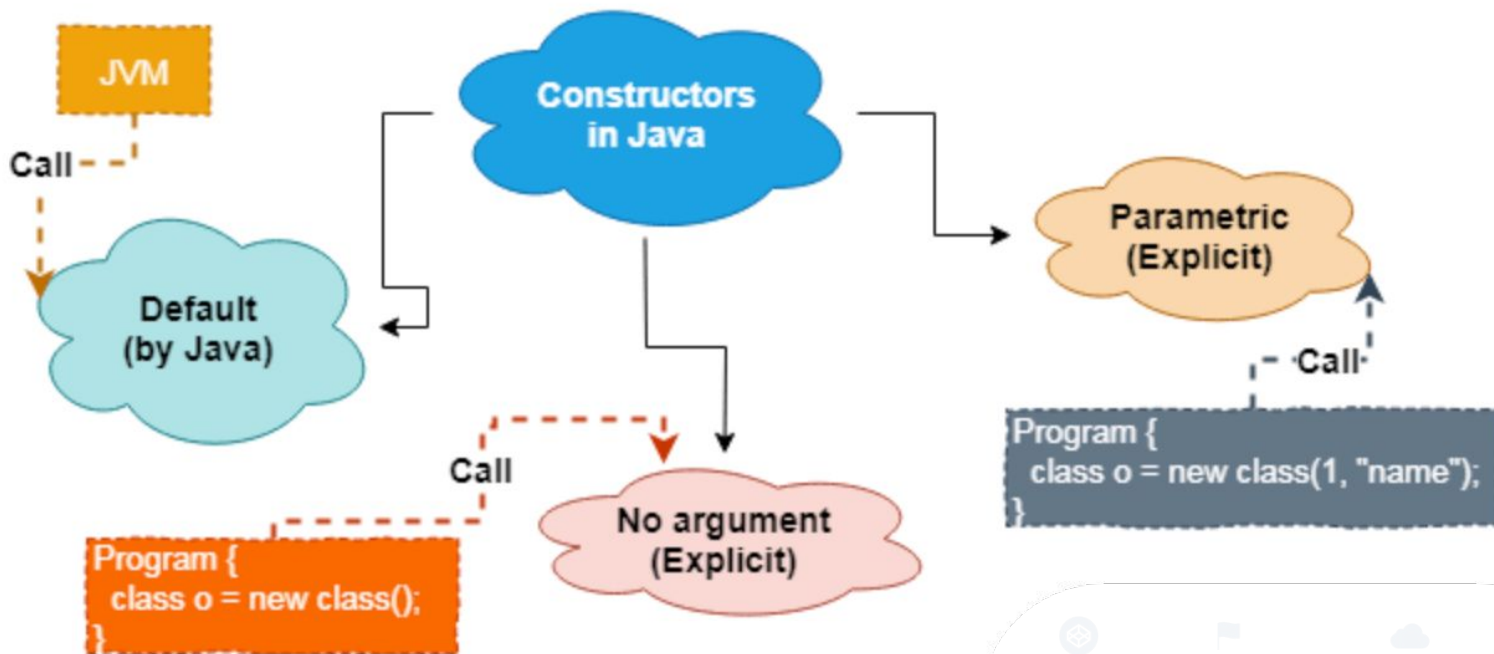
Під час передачі даних від одного кінця до іншого через мережу, вони повинні бути серіалізовані для успішного отримання даних, що означає перетворення в потік байтів перед надсиланням і перетворення його назад на кінці прийому. Щоб повідомити JVM про учасників, які не потребують серіалізації замість того, щоб бути втраченими під час передачі, з'являється тимчасовий модифікатор.



- Ініціалізація статичних полів та блоків виконується під час завантаження класу

- Ініціалізація нестатичних елементів виконується під час створення об'єкта(при виклику конструктора)

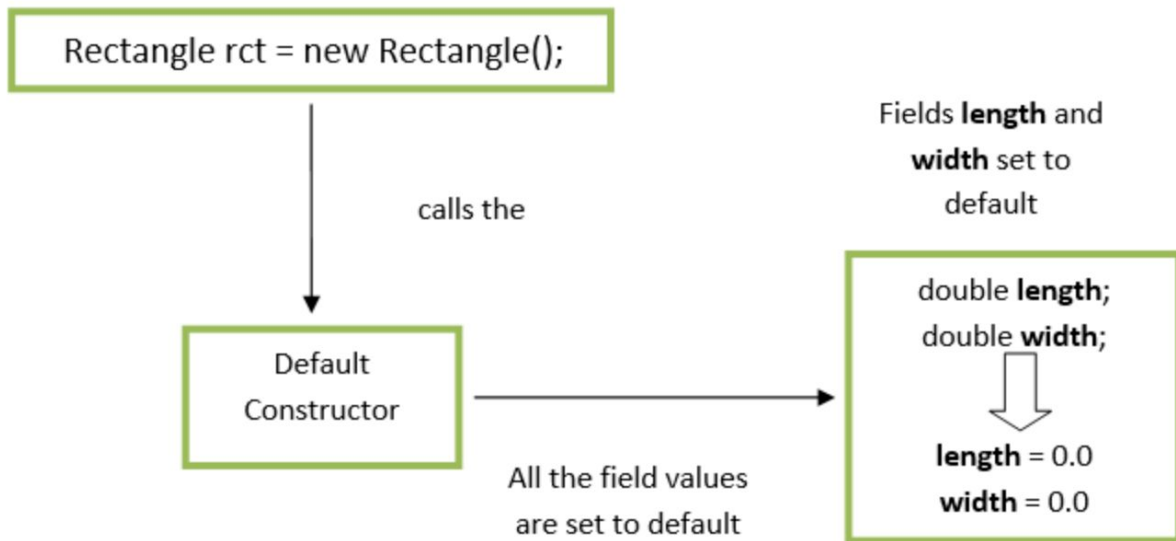
**Конструктор** – це метод, призначення якого полягає у створенні екземпляра класу.



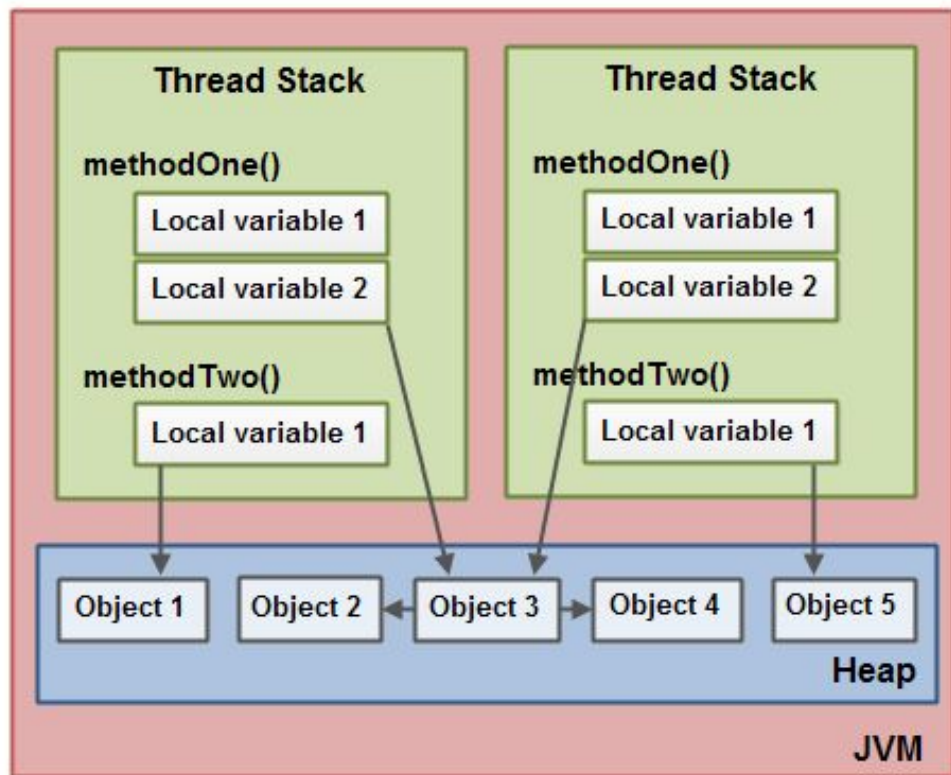


## Послідовність дій під час виклику конструктора

- Усі поля даних ініціалізуються своїми значеннями, передбаченими за промовчанням (0, false або null).
- Ініціалізатори всіх полів та блоки ініціалізації виконуються в порядку їх перерахування в оголошенні класу.
- Якщо в першому рядку конструктора викликається інший конструктор, то виконується викликаний конструктор.
- Виконується тіло конструктора

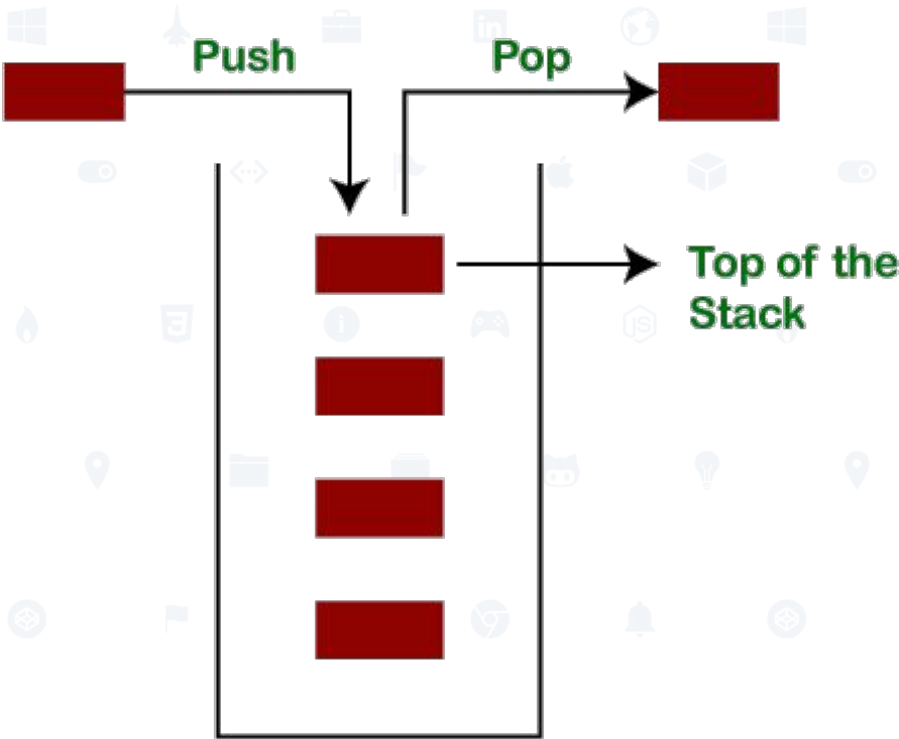



# Java Memory Model





# Stack





Стек працює за схемою LIFO (останнім увійшов, першим вийшов). Всякий раз, коли викликається новий метод, що містить примітивні значення або посилання на об'єкти, то на вершині стека під них виділяється блок пам'яті.

Коли метод завершує виконання, блок пам'яті, відведений для його потреб, очищається, і простір стає доступним для наступного методу

### **Основні особливості стеку**

Він заповнюється та звільняється у міру виклику та завершення нових методів

Змінні в стеку існують до тих пір, поки виконується метод у якому вони були створені

Якщо пам'ять стека буде заповнена, Java кине виняток `java.lang.StackOverFlowError`

Доступ до цієї області пам'яті здійснюється швидше, ніж до купи

Є потокобезпечним, оскільки для кожного потоку створюється свій окремий стек

boolean

byte

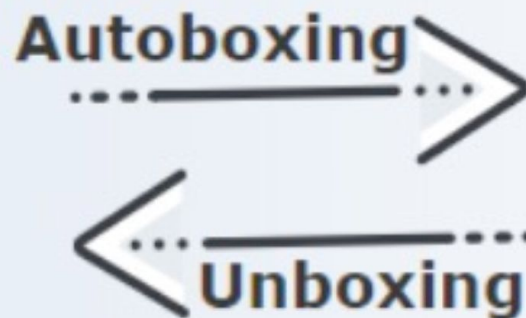
char

float

int

long

short



Boolean

Byte

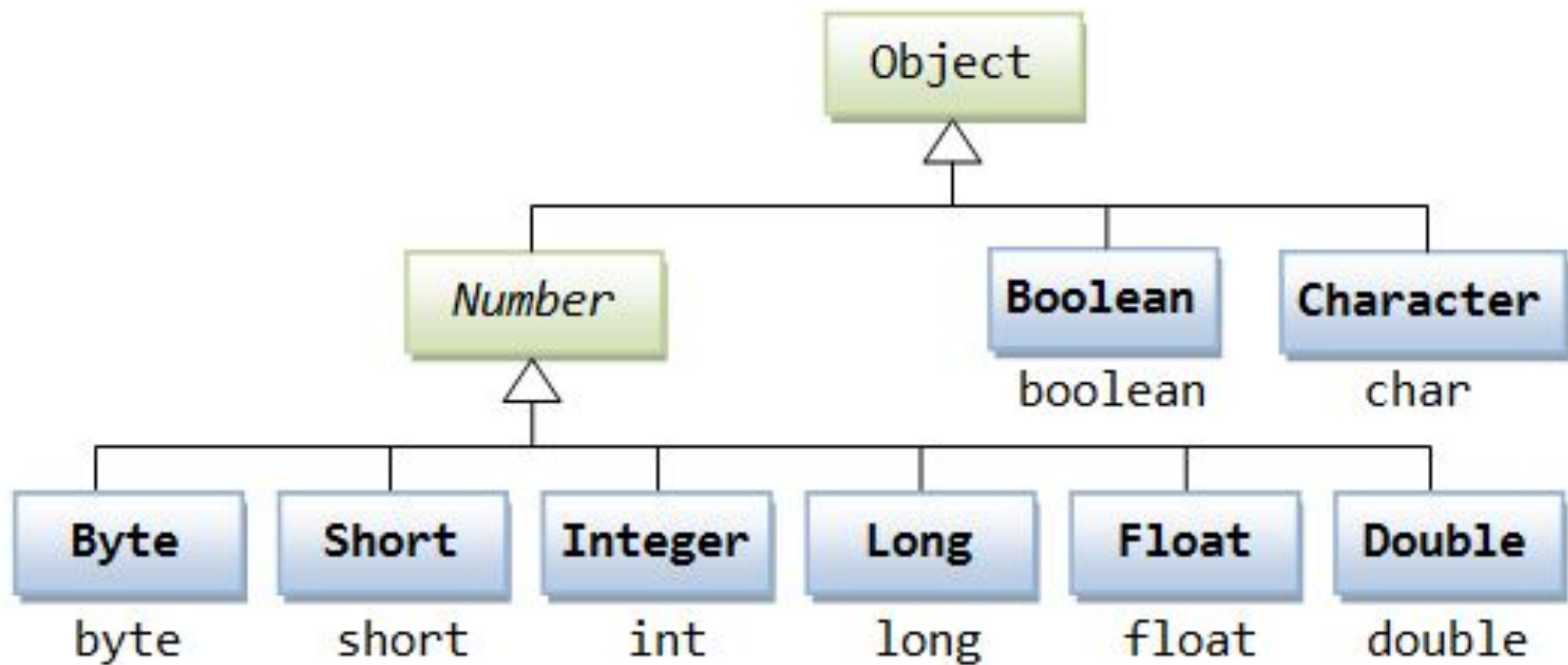
Character

Float

Integer

Long

Short



Integer i = 127

Integer j = 127

127

### Integer Constant pool

Integer i = new Integer(127)

127

Integer j = new Integer(127)

127

Integer i = 128

128

Integer j = 128

128

Integer i = 120

120

Integer j = new Integer(120)

120