**Unicode System, Naming Convention, Use of this and Super Keyword**

Computer systems internally store data in binary representation. A character is stored using a combination of 0's and 1's. The process is called **encoding**. A character encoding scheme is important because it helps to represent the same information on multiple types of devices.

**Types of Encoding**

Following are the different types of encoding used before the Unicode system.

1. ASCII (American Standard Code for Information Interchange): used for the United States
2. ISO 8859-1 used for the Western European Languages
3. KOI-8 used for Russian
4. GB18030 and BIG-5 used for Chinese and so on.
5. Base64 used for binary to text encoding

**Why does Java use Unicode System?**

There were a few limitations to the encoding techniques used before the Unicode system.

1. In every language, different letters are present and the code assigned to every letter is also different which means multiple languages have multiple codes for various letters.
2. Certain languages have many character sets, the code assigned to each character may vary in terms of length. For example, some character can be encoded with a single byte, other might require two or more bytes.

These problems led to finding a better solution for character encoding that is Unicode System.Play

## What is Unicode System?

* Unicode system is an international character encoding technique that can represent most of the languages around the world.
* Unicode System is established by Unicode Consortium.
* Hexadecimal values are used to represent Unicode characters.
* There are multiple Unicode Transformation Formats:
  1. UTF-8: It represents 8-bits (1 byte) long character encoding.
  2. UTF-16: It represents 16-bits (2 bytes) long character encoding
  3. UTF-32: It represents 32-bits (4 bytes) long character encoding.
* To access a Unicode character the format starts with an escape sequence \u followed by 4 digits hexadecimal value.
* A Unicode character has a range of possible values starting from \u0000 to \uFFFF.
* Some of the Unicode characters are  
  \u00A9 represent the copyright symbol - ©  
  \u0394 represent the capital Greek letter delta - Δ  
  \u0022 represent a double quote - "

### Program to convert UTF-8 to Unicode

**UnicodeDemo.java**

1. **public** **class** UnicodeDemo
2. {
3. **public** **static** **void** main(String ar[]) **throws** Exception
4. {
5. String str1 = "Unicode Sytem\u00A9";
6. **byte**[] charset = str1.getBytes("UTF-8");
7. String newstr = **new** String(charset, "UTF-8");
8. System.out.println(newstr);
9. }
10. }

**Output:**

Unicode Sytem©

In the above code, a Class **UnicodeDemo** is created. At the start, a Unicode String **str1** is converted into a UTF-8 form using the **getBytes()** method. After that, the byte array is again converted into Unicode and the value of **newstr** is displayed on the console.

### Problem Caused by Unicode

The Unicode standard was designed to represent 16-bit character encoding. But the 16-bits encoding was able to represent only 65,536 characters that were not sufficient for all the characters available around the world.

So, the Unicode system was extended up to 1,112,064 characters. The characters that are larger than 16-bits are called **Supplementary characters** and are defined by Java using a pair of char values.

**Naming Convention:**

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

## Advantage of Naming Conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Class

It should start with the uppercase letter.  
It should be a noun such as Color, Button, System, Thread, etc.  
Use appropriate words, instead of acronyms.

Public class Employee {

-----------------------

}

**Interface**

It should start with the uppercase letter.  
It should be an adjective such as Runnable, Remote, ActionListener.  
Use appropriate words, instead of acronyms.

interface **Printable {**

**}**

**Method**

It should start with lowercase letter.  
It should be a verb such as main(), print(), println().  
If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

Public class Employee {

Void draw() {

}

}

**Variable:**

It should start with a lowercase letter such as id, name.  
It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore).  
If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.  
Avoid using one-character variables such as x, y, z.

Public class Employee {

Int id;

}

**Package:**

It should be a lowercase letter such as java, lang.  
If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

//package  
package j**avaProjects;**  
class Employee {

-----------------------

}

**Constant**

It should be in **uppercase letters** such as RED, YELLOW.  
If the name contains multiple words, it should be separated by an underscore (\_) such as MAX\_PRIORITY.  
It may contain digits but not as the first letter.

class Employee {

static final int MIN\_AGE=18;

}

## CamelCase in Java naming conventions

Java follows **camel-case** syntax for naming the **class, interface, method, and variable.**

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, actionEvent, actionListener, etc.

Super Keyword in Java:

The**super** keyword in java is a reference variable that is used to refer to parent class objects.

It is majorly used in the following contexts:

* Use of super with variables
* Use of super with methods
* Use of super with constructors

## ****1. Use of super with variables****

When a derived class and base class has the same data members. In that case, there is a possibility of ambiguity for the JVM. At that time this Scenario Occurs, and if we want to inherit parent class member then using Super keyword which derive the base class properties even though if both are having same name.

Class Student {

Char grade = ‘A’;

}

Class Candidate extends Student {

Char grade = ‘B’;

 void display()

    {

        // print grade of base class (Student)

        System.out.println("Student grade: " + super.grade);

System.out.println("Student grade: " + grade);

    }

}

Class Test {

 public static void main(String[] args)

    {

        Candidate C = new Candidate();

        C.display();

    }

}

## ****2. Use of super with methods****

This is used when we want to call the parent class method. So whenever a parent and child class has the same-named methods then to resolve ambiguity we use the super keyword.

Class Person {

    void message()

    {

        System.out.println("This is Person class");

}

}

class Student extends Person {

    void message()

    {

        System.out.println("This is student class");

Super.message();

    }

    // Note that display() is

    // only in Student class

    void display()

    {

        // will invoke or call current

        // class message() method

        message();

        // will invoke or call parent

        // class message() method

        super.message();

    }

}

// Driver Program

class Test {

    public static void main(String args[])

    {

        Student s = new Student();

        // calling display() of Student

        //s.display();

s.message();

    }

}

// superclass Person

class Person {

    Person()

    {

        System.out.println("Person class Constructor");

    }

}

// subclass Student extending the Person class

class Student extends Person {

    Student()

    {

        // invoke or call parent class constructor

        super();

        System.out.println("Student class Constructor");

    }

}

// Driver Program

class Test {

    public static void main(String[] args)

    {

        Student s = new Student();

    }

}

**Note:**

* Call to super() must be the first statement in the Derived(Student) Class constructor because if you think about it, it makes sense that the superclass has no knowledge of any subclass, so any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it needs to complete its execution first.
* If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. The object *does* have such a constructor, so if the Object is the only superclass, there is no problem.
* If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors is called, all the way back to the constructor of Object. This, in fact, is the case. It is called [*constructor chaining*](https://www.geeksforgeeks.org/constructor-chaining-java-examples/)*.*

# Constructor Chaining

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining can be done in two ways: 

* **Within same class**: It can be done using **this()** keyword for constructors in the same class
* **From base class:**by using **super()** keyword to call the constructor from the base class.

**Why do we need constructor chaining?**

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

class Temp

{

    // default constructor 1

    // default constructor will call another constructor

    // using this keyword from same class

    Temp()

    {

        // calls constructor 2

        this(5);

        System.out.println("The Default constructor");

    }

    // parameterized constructor 2

    Temp(int x)

    {

        // calls constructor 3

        this(5, 15);

        System.out.println(x);

    }

    // parameterized constructor 3

    Temp(int x, int y)

    {

        System.out.println(x \* y);

    }

    public static void main(String args[])

    {

        // invokes default constructor first

        new Temp();

    }

}

**Output:**

75

5

The Default constructor

**Rules of constructor chaining :**

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

// Java program to illustrate Constructor Chaining

// within same class Using this() keyword

// and changing order of constructors

class Temp

{

// default constructor 1

Temp()

{

System.out.println("default");

}

// parameterized constructor 2

Temp(int x)

{

// invokes default constructor

this();

System.out.println(x);

}

// parameterized constructor 3

Temp(int x, int y)

{

// invokes parameterized constructor 2

this(5);

System.out.println(x \* y);

}

public static void main(String args[])

{

// invokes parameterized constructor 3

new Temp(8, 10);

}

}

**Alternative method : using Init block**:   
When we want certain common resources to be executed with every constructor we can put the code in the **Init block**. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

class Temp

{

// block to be executed before any constructor.

{

System.out.println("init block");

}

// no-arg constructor

Temp()

{

System.out.println("default");

}

// constructor with one argument.

Temp(int x)

{

System.out.println(x);

}

public static void main(String[] args)

{

// Object creation by calling no-argument

// constructor.

new Temp();

// Object creation by calling parameterized

// constructor with one parameter.

new Temp(10);

}

}

### ****Similarities in this and super****

* We can use *this* as well as *super* **anywhere except static area**. Example of this is already shown above where we use this as well as super inside public static void main(String[]args) hence we get Compile Time Error since cannot use them inside static area.
* We can use *this* as well as *super* **any number of times in a program**.
* Both are **non-static** keyword.

// Java Program to illustrate using this

// many number of times

class RRR {

// instance variable

int a = 10;

// static variable

static int b = 20;

void GFG()

{

// referring current class(i.e, class RR)

// instance variable(i.e, a)

this.a = 100;

System.out.println(a);

// referring current class(i.e, class RR)

// static variable(i.e, b)

this.b = 600;

System.out.println(b);

// referring current class(i.e, class RR)

// instance variable(i.e, a) again

this.a = 9000;

System.out.println(a);

}

public static void main(String[] args)

{

new RRR().GFG();

}

}