# java.lang.Object class

eazy bytes

In Java, there's a special class called **Object** that belongs to the **java.lang** package and is part of the fundamental **java.base** module.

Every class you use or create in Java is connected to this Object class in some way. Think of it like a big family tree: all classes are like children, grandchildren, or even great-grandchildren of the Object class.

In simpler terms, the Object class is like the ultimate ancestor for all classes in Java. It's at the top of the class hierarchy, and while other classes can have parents, **the Object class doesn't have a parent itself.**

Since Object is the super class for all the Java classes in Java, a reference variable of the **Object** class can be used to hold a reference of an object of any class. For example, all the below assignments are valid,

```
Object obj;
obj = null;
obj = new Object();
Person person = new Person();
obj = person;
obj = new AnyClass();
```

# java.lang.Object class

eazy bytes

The opposite assignment is not valid. We cannot assign a reference of an object of the Object class to a reference variable of any other type.

```
Person person = new Object(); // compilation error
```

Sometimes, you might put a special kind of thing, like an ”Person," inside a box that's labeled as a general "Object" box. Later, when you want to take it out and use it as the special ”Person" again, you have to do something called a **"casting”**. It's like telling the computer, "Hey, remember that thing in the 'Object' box? It's actually an Person,' so let me treat it like one."

```
Object obj2 = new Person();
Person person = (Person) obj2; // downcasting
```

It's important to note that if the actual type of the object referred to by obj2 is not compatible with Person, a **ClassCastException** will occur at runtime. Therefore, it's generally a good practice to check the type before casting using **instanceof** to avoid such exceptions:

```
if (obj2 instanceof Person) {
    Person person = (Person) obj2;
    // Now you can safely use 'act' as an Account
} else {
    // Handle the case where obj2 is not an instance of Account
}
```

# java.lang.Object class

eazy bytes

The **Object** class in Java has **nine** methods that all classes can use. These methods fall into two categories.

### final methods

The first category includes below methods. You must use them as they are; you can't change how they work in your own classes.

- `getClass()`
- `notify()`
- `notifyAll()`
- `wait()` (overloaded methods)

### non final methods

The second category includes below methods.You can customize how these methods work in your classes by overriding their implementation.

- `toString()`
- `equals()`
- `hashCode()`
- `clone()`
- `finalize()`

# java.lang.Object class

eazy bytes

| Method name | Implemented ? | Overriden ? | Description |
|---|---|---|---|
| hashCode | ✓ | ✓ | Returns a hash code value of an object as an int value |
| equals | ✓ | ✓ | Used to compare two objects for equality |
| toString | ✓ | ✓ | Returns a string representation of an object |
| finalize | ✗ | ✓ | Invoked by the garbage collector before an object is destroyed. Deprecated from Java 9 |
| clone | ✗ | ✓ | Used to make a copy or clone of an object |
| getClass | ✓ | ✗ | Returns the runtime class of this Object |
| notify | ✓ | ✗ | Wakes up a single thread that is waiting on this object's monitor |
| notifyAll | ✓ | ✗ | Wakes up all threads that are waiting on this object's monitor |
| wait (overloaded methods) | ✓ | ✗ | Makes a thread wait in the wait queue of the object with or without a timeout |

# Object.getClass()

eazy bytes

The primary purpose of the **getClass()** method is to retrieve the runtime class of an object. It returns an object of type **Class<?>**, which represents the class to which the object belongs. It's commonly used to determine the actual type of an object at runtime.

Here is the getClass() method signature inside Object class,

```
public final native Class<?> getClass();
```

In Java, the **native** keyword indicates that the implementation of a method is provided by a platform-dependent native code, typically written in languages like C or C++. The method's body is not written in Java but is implemented externally, usually to interact with the underlying system or perform tasks that are not possible or efficient to implement in Java.

In the case of the getClass() method, the native keyword suggests that its implementation is not written in Java but is instead provided by the Java Virtual Machine (JVM) itself or by a specific platform-dependent library. This is because retrieving the runtime class of an object involves interacting with low-level details of the JVM or the underlying system.

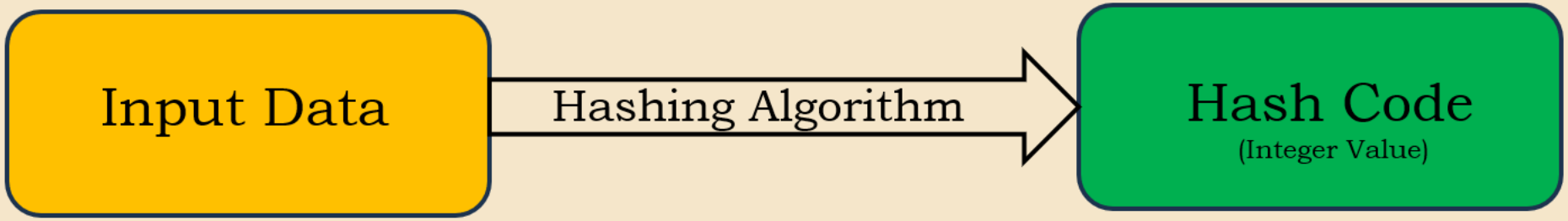
Example code,

```
Person person = new Person();
Class prsnClass = person.getClass();
System.out.println(prsnClass.getName());
System.out.println(prsnClass.getSimpleName());
System.out.println(prsnClass.getPackageName());
```

# Object.hashCode()

eazy bytes

A hash code is like a special number that is calculated for some information using a specific method, which is called a hash function. People might also call it a hash sum or a hash value. It's a way of turning information into a unique number.




Computing a hash code is a **one-way process**. When you create a hash code for something, it's like turning it into a secret code that's hard to turn back into the original thing. Figuring out the original information from the hash code is not easy, and that's not the point of making a hash code in the first place.

The Object class contains a hashCode() method that returns an int, which is the hash code of the object. **The default implementation of this method computes the hash code of an object by converting the memory address of the object into an integer.** Since the hashCode() method is defined in the Object class, it is available in all classes in Java. However, you are free to override the implementation in your class.

# Object.hashCode()

eazy bytes

## Why would we need a hashCode value of a object in Java ?

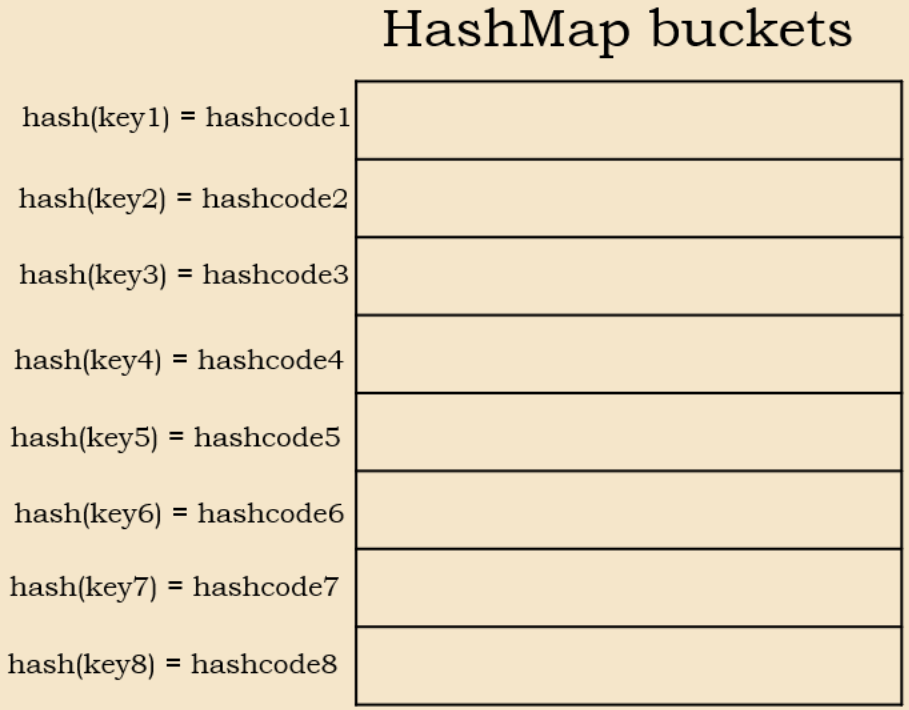
A hash code is like a label that helps organize and find information quickly. When we put information in a special storage (like a box with many compartments), we calculate its hash code and put it in a specific compartment based on that code. Later, when we want to find that information, we use its hash code to go directly to the right compartment and get it faster. So, a hash code makes storing and finding data in collections quicker and more efficient.

The primary purposes of the hashCode() method are related to hash-based data structures, such as hash tables, which are widely used in Java collections like **HashMap, HashSet**, and others.

The HashMap is a part of Java's **collections** framework. When we add a **(key, value)** pair to a HashMap, it calculates the key's hash code, and this hash code becomes the bucket number.

Keys that produce the same hash code are grouped into the same bucket, forming a linked list where multiple (key, value) pairs can be stored together. However, keys with different hash codes end up in different buckets.

When we want to access the value associated with a specific key, the key's hash code is recalculated. This hash code guides us to the corresponding bucket, and from there, we retrieve the associated value. This process allows for efficient storage and retrieval of key-value pairs in a HashMap.



# Object.hashCode()

eazy bytes

Here are the rules that must be followed while overriding the hashCode() method in your class. Suppose there are two object references, obj1 and obj2:

1) **Equal Objects Should Have the Same Hash Code:** If obj1.equals(obj2) returns true, then obj1.hashCode() must return an integer, which is equal to obj2.hashCode(). That is, if two objects are equal using the equals() method, they must have the same hash code.

2) **Equal Hash Codes Don't Guarantee Equality:** If obj1.hashCode() is equal to obj2.hashCode(), it is not guarantee that obj1.equals(obj2) returns true. That is, if two objects have the same hash code using the hashCode() method, they do not have to be equal using the equals() method.

3) **Consistency of Hash Code:** If the hashCode() method is invoked on the same object multiple times in the same execution of a Java application, the method must return the same hash code (integer) value.

4) **Tying Hash Code and Equality:** The hashCode() and equals() methods are closely tied. If your class overrides any of these two methods, it must override both for the objects of your class to work correctly in hash-based collections. Another rule is that you should use only those instance variables to compute the hash code for an object, which are also used in the equals() method to check for equality.

In Java 7, a utility class called **java.lang.Objects** was introduced. This class includes a **hash()** method capable of calculating the hash code for multiple values of any type. It is recommended to utilize the **Objects.hash()** method for computing the hash code of an object.

# Object.equals()

eazy bytes

Just like how identical twins share a strong physical resemblance, the **equals** method is used to compare the content or attributes of two objects in Java. It determines whether two objects are similar in their "attributes."

Below is the default equals method logic provided by Object class. It takes an object as an argument and returns a boolean value indicating whether the current object is equal to the specified object.

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

The default implementation of equals in the Object class compares object references for equality. In other words, it checks if the two objects being compared refer to the exact same memory location.

It's common for classes to override the default equals method to provide a more meaningful comparison based on the content of the objects.

# Object.equals()

eazy bytes

```java
public class Person {

    private String name;
    private int age;
    private char gender;
    private int ssn;

    // Constructor, Getter, Setter methods

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && gender == person.gender && ssn == person.ssn
                && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, ssn);
    }
}
```

Attached is the sample code to override hashCode and equals method inside a Person class.

Since we have implemented hashCode and equals method based on the Person instance fields like name, age, gender & ssn, the hashCode values of two different person objects with same attributes are calculated as same value.

The same instance fields should also consider as part of the equals method logic as well.

```java
Person p1 = new Person("John", 25, 'M', 564323456);
Person p2 = new Person("John", 25, 'M', 564323456);
System.out.println(p1.hashCode()); //798228002
System.out.println(p2.hashCode()); //798228002
System.out.println(p1.equals(p2)); // true
```

# Object.equals()

eazy bytes

Below are the guidelines for implementing equals:

**1) Symmetry:** The equals method should be symmetric, meaning if a.equals(b) is true, then b.equals(a) should also be true.

**2) Reflexivity:** The equals method should be reflexive, meaning a.equals(a) should always be true.

**3) Transitivity:** The equals method should be transitive, meaning if a.equals(b) is true and b.equals(c) is true, then a.equals(c) should also be true.

**4) Consistency:** The equals method should provide consistent results over multiple invocations as long as the objects are not modified.

**5) Handling null:** It's common to check for null to avoid NullPointerExceptions in the equals method.

**6) Relationship with the hashCode() method:** In the realm of Java object comparison, there exists a connection between the equals() and hashCode() methods. When a.equals(b) evaluates to true, it's imperative that a.hashCode() produces the same value as b.hashCode(). **In simpler terms, if two objects are considered equal by the equals() method, their respective hash code values, as retrieved from hashCode(), must match.**

However, it's essential to note that the reverse does not necessarily hold true. If two objects share the same hash code, it doesn't guarantee equality based on the equals() method. In other words, if a.hashCode() equals b.hashCode(), it does not imply that a.equals(b) will also be true. **The equality of hash codes does not automatically ensure the equality of the objects themselves.**

# Object.toString()

eazy bytes

The primary purpose of the **toString()** method is to return a string representation of an object. This string should be concise and informative, conveying relevant information about the object's state.

Below is the default toString method logic provided by Object class. It returns a string in the format **<fully-qualified-class-name>@<hash-code-of-object-in-hexadecimal-format>**

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

It's common for classes to override the toString() method to provide a customized string representation based on the object's attributes. The toString() method typically formats the string to include the names and values of the object's attributes. This makes it easier for developers to understand the object's state when logging or debugging.

The toString() method should include all relevant attributes to provide a comprehensive view of the object's state.
Avoid including sensitive information in the toString() output, especially if the class is part of a public API.

In summary, the toString() method in Java is used to provide a human-readable string representation of an object's state. Overriding this method allows developers to customize the format of the string output, making it more informative and suitable for debugging and logging purposes.

# Object.toString()

eazy bytes

```java
public class Person {

    private String name;
    private int age;
    private char gender;
    private int ssn;

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender=" + gender +
                ", ssn=" + ssn +
                '}';
    }
}
```

**In this implementation:**

- The toString() method includes the names and values of all the attributes (name, age, gender, and ssn).
- It follows the common pattern of formatting the output as "ClassName{attribute1=value1, attribute2=value2, ...}".
- Adjustments can be made based on specific formatting preferences or additional considerations for your use case.

```java
Person p1 = new Person("John", 25, 'M', 564323456);
System.out.println(p1);
// Person{name='John', age=25, gender=M, ssn=564323456}
```

# Object.finalize()

eazy bytes

In Java, when an object consumes resources that require release when it's no longer needed, the language offers a mechanism for performing cleanup or resource release just before the object is discarded. While you can create objects in Java, the manual destruction of objects isn't allowed. Instead, the Java Virtual Machine (JVM) employs a low-priority process known as the garbage collector to identify and eliminate objects that are no longer referenced.

This garbage collector provides an opportunity to execute cleanup code before an object is effectively destroyed. The Object class includes a method named **finalize()** for this purpose, declared as follows:

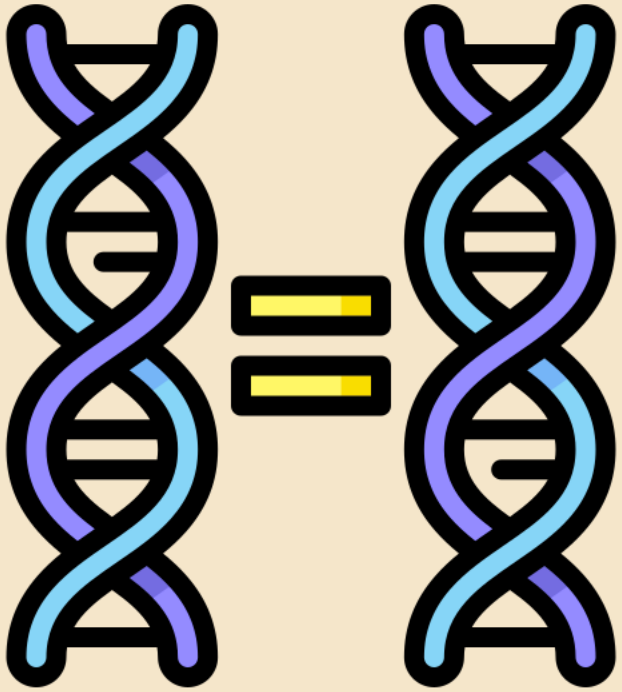
```
protected void finalize() throws Throwable {

}
```

By default, the finalize() method in the Object class does not do anything. You need to override the method in your class. The finalize() method of your class will be called by the garbage collector before an object of your class is destroyed. But it is not guaranteed that a finalizer will run at all. This makes it undependable for a programmer to write cleanup logic in the finalize() method.

The finalize() method in the Object class has been deprecated since Java 9 because using the finalize() method to clean up resources is inherently problematic. There are several better alternatives to clean up resources, for example, using try-with-resources and try-finally blocks.

# Object.clone()

eazy bytes



In Java, there isn't an automatic mechanism for cloning (creating a copy) of objects. When you assign one reference variable to another, only the reference to the object is copied, not the actual content of the object. Cloning involves making a copy of the object's content bit by bit.

To enable cloning for objects of your class, you need to manually reimplement the **clone()** method within your class. Once the clone() method is appropriately overridden, you can then clone objects of your class by invoking the clone() method on them.

Below is the default clone method logic provided by Object class

```
protected native Object clone() throws CloneNotSupportedException;
```

The clone() method in the Object class has all the code that is needed to clone an object in native code format. All you need to do is invoke it from the clone() method of your class. It will make a bitwise copy of the original object and return the reference of the copy.

It is crucial to implements **Cloneable** marker interface in your class if you intend to enable cloning. otherwise, the clone() method in the Object class will throw a **CloneNotSupportedException**. This implies that when invoking the clone() method of the Object class, it is necessary to either enclose the call within a try-catch block or propagate the exception using throws.

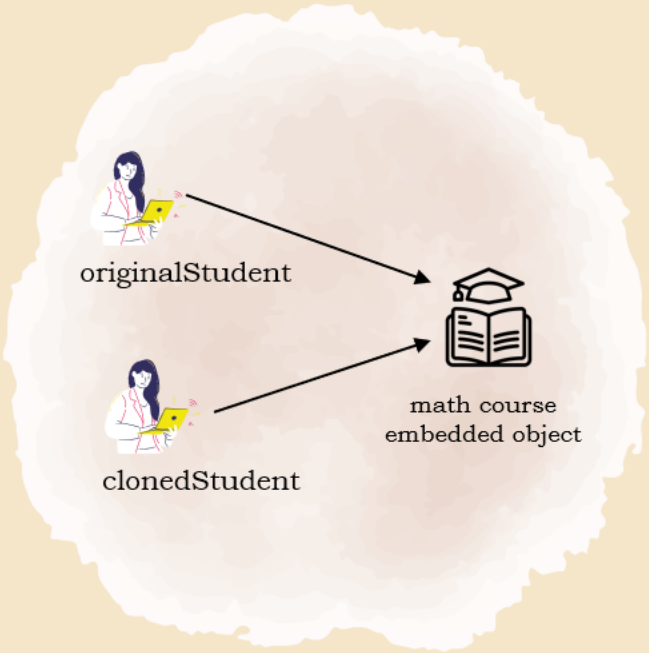
# Object.clone()

eazy bytes

There are two types of cloning, **1) Shallow Cloning  2) Deep Cloning**

**Shallow cloning** creates a new object but does not create copies of the objects referenced by the original object. Instead, it copies references to those objects. When you clone the Student object, the reference of the course object is cloned. After cloning is performed, there are two copies of the Student object; both of them have references to the same course object. This is called a shallow cloning because references are copied, not the objects. The clone() method of the Object class makes only shallow cloning, unless you code it otherwise.

```java
class Student implements Cloneable {
    private String name;
    private Course course;

    public Student(String name, Course course) {
        this.name = name;
        this.course = course;
    }

    // Getters and setters...

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Course {
    private String courseName;

    // Getters and setters...
}
```




Heap memory

# Object.clone()

eazy bytes

Below is the sample shallow cloning example of Student object,

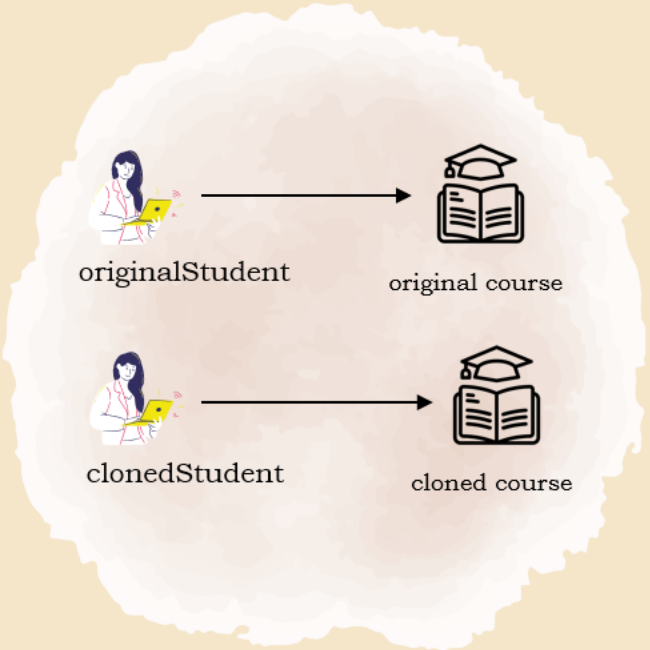
```
public class ShallowCloneExample {
    public static void main(String[] args) throws CloneNotSupportedException {
        Course math = new Course();
        math.setCourseName("Math");

        Student originalStudent = new Student("Alice", math);
        Student clonedStudent = (Student) originalStudent.clone();

        System.out.println(originalStudent == clonedStudent); // false
        System.out.println(originalStudent.getCourse() == clonedStudent.getCourse());
    }
}
```

# Object.clone()

eazy bytes

**Deep cloning** creates a new object and also creates copies of the objects referenced by the original object. It ensures that nested objects are cloned recursively. The nested objects within the class must also implement the Cloneable interface.

```
class Student implements Cloneable {
    private String name;
    private Course course;

    public Student(String name, Course course) {
        this.name = name;
        this.course = course;
    }

    // Getters and setters...

    @Override
    public Object clone() throws CloneNotSupportedException {
        Student clonedStudent = (Student) super.clone();
        clonedStudent.course = (Course) course.clone(); // deep cloning the nested object
        return clonedStudent;
    }
}

class Course implements Cloneable {
    private String courseName;

    // Getters and setters...

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```



# Object.clone()

eazy bytes

Below is the sample deep cloning example of Student object,

```
public class DeepCloneExample {
    public static void main(String[] args) throws CloneNotSupportedException {
        Course math = new Course();
        math.setCourseName("Math");

        Student originalStudent = new Student("Alice", math);
        Student clonedStudent = (Student) originalStudent.clone();

        System.out.println(originalStudent == clonedStudent); // false
        System.out.println(originalStudent.getCourse() == clonedStudent.getCourse()); // false
    }
}
```

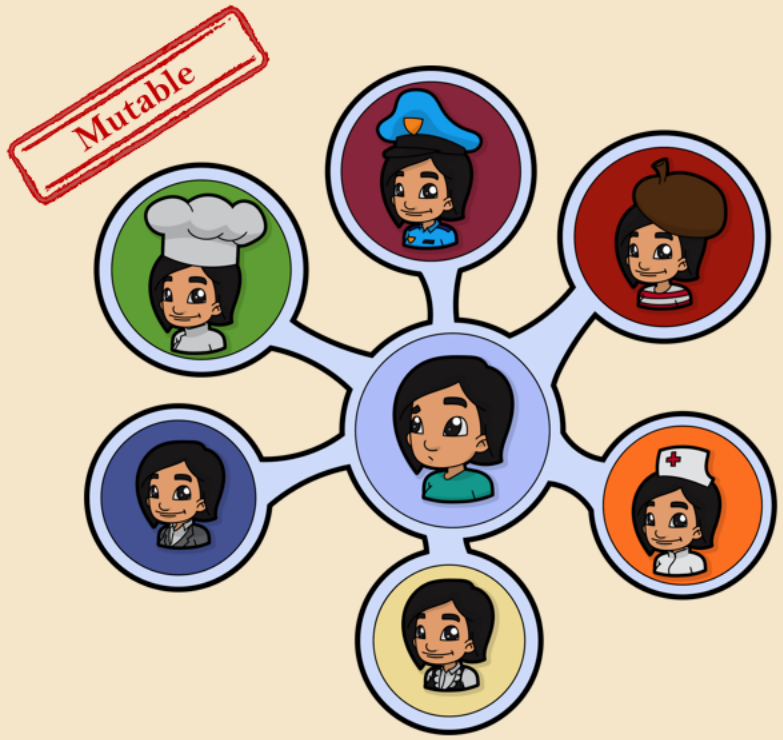
# Mutable & Immutable objects

eazy bytes

In Java, **mutable** and **immutable** are terms used to describe the ability of an object to be modified after it has been created.

**Immutable** objects are objects whose state **cannot be changed** after they are created. Once an immutable object is created, its state cannot be changed. If you need to modify an immutable object, you must create a new instance of the object with the modified state.

**Mutable** objects, on the other hand, are objects whose state **can be changed** after they are created. With mutable objects, you can add, remove or modify elements of the object without creating a new instance.




You can create an instance of this Person class and modify its properties as follows:

```
Person person = new Person("Elma Raymond", "Software Engineer");

person.setOccupation("Chef");
```

As you can see, the setName and setOccupation methods can be used to modify the values of the name and occupation properties of the Person object. This makes the Person class mutable.

## Sample mutable Person class

```
public class Person {

private String name;
private String occupation;

public Person(String name, String occupation) {
  this.name = name;
  this.occupation = occupation;
}

public String getName() {
   return name;
}
public String getOccupation() {
   return occupation;
}
public void setName(String name) {
   this.name = name;
}
public void setOccupation(String occupation) {
   this.occupation = occupation;
}

}
```

# How to create immutable objects ?

eazy bytes

To make the Person class immutable, you need to **remove the setters**, which allow the properties of the object to be modified after creation and declare properties as **final**. Here's an updated version of the Person class that is immutable:


```
Person person = new Person("Chitra Mansi", "Doctor");
```

By removing the setters, the Person class is now immutable, meaning that it cannot be modified after it is created. If you need to create a new Person object with different values, you must create a new instance of the class.

Note that it's also a good practice to mark the class itself as final to prevent subclasses from modifying the behavior of the class.

Immutable class is also known as **data carrier class**.A data carrier class is a class that is primarily used to store data and provide methods to access that data. These classes are often used to transfer data between different parts of a program, such as between the front-end and back-end of an application or between different layers of a system.

## Sample immutable Person class

```
public final class Person {

private final String name;
private final String occupation;

public Person(String name, String occupation) {
  this.name = name;
  this.occupation = occupation;
}

public String getName() {
   return name;
}
public String getOccupation() {
   return occupation;
}

}
```

eazy bytes

# Mutable

Mutable objects require synchronization when accessed by multiple threads to avoid issues like race conditions, where two threads may attempt to modify the same object simultaneously, leading to inconsistent state.

Mutable objects are more flexible and efficient, but require careful handling to avoid issues related to concurrent access.


# Immutable

Immutable objects are thread-safe, meaning that they can be accessed by multiple threads without the need for synchronization. This is because the state of an immutable object cannot be changed after it is created.

May require more memory overhead and potentially slower performance due to the need to create new instances.

⚠ Overall, the choice between mutable and immutable objects in Java depends on the specific use case and performance requirements of your application.

From Java 16

# Record classes

eazy bytes

Record classes are a new feature introduced in Java 16 that provide a **concise way** to declare classes whose main purpose is to carry/store the data. They are similar to traditional classes, but are designed to be immutable and provide default implementations for methods like **equals**, **hashCode**, and **toString**. Here's an example/syntax of a record class in Java:

```
public record Person(String name, String occupation) {

}
```

The code that's in parentheses, is called the **record header** & it consists of **record components**, a comma delimited list of components.

In this example, the Person class is declared as a record by using the **record** keyword. The class has two properties: name, occupation which are of type string. The record class automatically generates the following methods:

- ✓ **Constructor**: A constructor that takes the same number of arguments as the number of properties.
- ✓ **Getters**: A getter method for each property. The method name will not have a prefix get as they are no setter methods.
- ✓ **equals():** An implementation of the equals() method that compares the values of the properties.
- ✓ **hashCode():** An implementation of the hashCode() method that uses the values of the properties.
- ✓ **toString():** An implementation of the toString() method that returns a string representation of the object.

With a record class, you can create instances of the class using the constructor and access the properties using the getter methods. You can also compare record objects using the equals() method, which compares the values of the properties. Here's an example:

```
Person person = new Person("Chitra Mansi", "Doctor");
System.out.println(person.name());
System.out.println(person.occupation());
```

**Overall, record classes provide a convenient way to declare immutable data classes in Java with a minimal amount of boilerplate code.**

From Java 16

# Record classes

eazy bytes

## The Canonical Constructor of a Record Class

The following example explicitly declares the **canonical constructor** for the Person record class. It verifies that name and occupation are not null. If they are null, it throws an IllegalArgumentException:

```
record Person(String name, String occupation) {

 public Person(String name, String occupation) {
     if (name == null || occupation== null) {
         throw new java.lang.IllegalArgumentException();
     }
     this.name = name;
     this.occupation = occupation;
 }

}
```

Repeating the record class's components in the signature of the canonical constructor can be tiresome and error-prone. To avoid this, you can declare a **compact constructor** whose signature is implicit (derived from the components automatically). For example, the following compact constructor declaration validates name and occupation in the same way as in the previous example:

```
public Person {
    if (name == null || occupation== null) {
        throw new java.lang.IllegalArgumentException();
    }
}
```

This compact form of constructor declaration is only available in a record class. Note that the statements this.name = name; and this.occupation = occupation; which appear in the canonical constructor do not appear in the compact constructor. At the end of a compact constructor, its implicit formal parameters are assigned to the record class's private fields corresponding to its components.


# Record classes

eazy bytes

- You can declare static fields, static initializers, and static methods in a record class, and they behave as they would in a normal class

- You cannot declare instance variables (non-static fields) or instance initializers in a record class, but instance methods are allowed

- A record class is implicitly final, so you cannot explicitly extend a record class. However, beyond these restrictions, record classes behave like normal classes

- You can declare a record class that implements one or more interfaces. The abstract class java.lang.Record is the common superclass of all record classes. Due to this reason, we can't extend another class inside a record class

- The class **java.lang.Class** has two new methods related to record classes:

  **RecordComponent[] getRecordComponents():** Returns an array of java.lang.reflect.RecordComponent objects, which correspond to the record class's components.

  **boolean isRecord():** Returns true if the class was declared as a record class.

From Java 10

# var (local variable type inference)

eazy bytes

Java 10 introduced a new way to declare variables using the **var** keyword. It allows you to declare local variables without explicitly specifying their type. Instead, the type of the variable is inferred by the compiler based on the expression used to initialize it.

For example, instead of writing:

```
String message = "Hello, World!";
```

you can use var to declare the variable without specifying its type. The compiler will infer that the type of message is String, based on the expression "Hello, World!". Usage of var helps with increasing the readability of the code, and to reduce boilerplate code.

```
var message = "Hello, World!";
```

var is a reserved identifier name, not a keyword, which means that existing code that uses var as a variable, method, or package name is not affected. However, code that uses var as a class or interface name is affected and the class or interface needs to be renamed.

Due to this automatic inference of the type of a variable, it is always necessary to assign an initial value to the variable. For example, the following is not allowed:

```
var message;    // not allowed
```

From Java 10

# var (local variable type inference)

eazy bytes

Using var can help avoid repetition and encourage DRY (Don't Repeat Yourself) coding practices. For example,

```
MyCustomClass obj = new MyCustomClass(); -> var obj = new MyCustomClass();
```

More examples on the usage of var,

```
var nums = new int[] {1,2,3,4,5}; // infers Array

for(var num: nums) {   // Enhanced for loop
    System.out.println(num);
}

for (var counter = 0; counter < 10; counter++)   {...}    // for loop which infers int
```

```
var number = 16;// compiler interprets number as int
number = "32"; // This will result in compilation error because you are trying to assign String to int type
```

It's worth noting that while JavaScript also has a var keyword, it functions differently from the Java 10 var. In JavaScript, there are no type definitions for variables, so the example shown earlier would have been interpreted successfully by the JavaScript runtime. This difference between the two languages is one of the reasons why TypeScript was introduced, to provide a stronger typing system for JavaScript code.

From Java 10

# `var` (local variable type inference)

eazy bytes

Consider an **inheritance scenario** where there are two subclasses, **Student** and **Artist**, extended from the parent class **Person**. Suppose someone creates an object of the Student class using the var keyword, as shown below:

```
var s = new Student(); // In this case, is the type of s Student or Person?
```

In this case, the type of s is Student, since it is initialized with an object of the Student class. Note that when a variable is declared with the var keyword, it takes on the type of the initializer.

So, if you try to reassign the p variable to an object of the Engineer class, as shown below:

```
s = new Artist(); // Compilation error: incompatible types
```

It will result in a compilation error, stating that the types are incompatible. This is because once a variable is initialized with a specific type, its type cannot be changed to a different type.

From Java 10

# var limitations

eazy bytes

We can't use var with method arguments:

```
public int sum(var a, var b) { // invalid

}
```

We can't use var for method return types,

```
public var sum(int a, int b) { // invalid
    return a + b;
}
```

You cannot initialize a var variable to null. By assigning null, it is not clear what the type should be, since in Java, any object reference can be null.

```
var count=null;// invalid
```

We can't use var for field declarations on a class.

```
public class Example {
    var String = "Hello"; // invalid
}
```

var can't be used without an assignment,

```
var name; // invalid
```