Java Classes (Definition and use)

Declaring new classes

A new class is declared with the `class` keyword followed by the name of the class.

```
class Nothing {
    // empty body
}
```

A class body can include fields, methods, and constructors. **Fields** store data, **methods** define behavior and **constructors** allow us to create and initialize new objects of the class.

A **field** is a variable that stores data. It may have any type, including primitive types (int, float, boolean and so on) and classes (even the same class). A class can have as many fields as you need.

Let's declare a class `Patient`:

```java
/**
 * The class is a "blueprint" for patients
 */
class Patient {
    String name;
    int age;
    float height;
}
```

This class represents a patient in a hospital information system. It has three fields for storing important information about the patient: `name`, `age`, and `height`. All objects of the class `Patient` have the same fields, but their values may be different for each object.

Creating objects

Let's create an **instance** of the class `Patient` using the keyword **new**:

```java
Patient patient = new Patient();
```

When you create a new object, each field is initialized with the default value of the corresponding type.

```java
System.out.println(patient.name); // it prints null
System.out.println(patient.age);  // it prints 0
```

The following program creates two objects of the class `Patient` and prints the information about them.

```
public class PatientDemo {

    public static void main(String[] args) {

        Patient john = new Patient();

        john.name = "John";
        john.age = 30;
        john.height = 180;

        System.out.println(john.name + " " + john.age + " " + john.h

        Patient alice = new Patient();

        alice.name = "Alice";
        alice.age = 22;
        alice.height = 165;

        System.out.println(alice.name + " " + alice.age + " " + alic
    }
}
```

So, the output of the code above is:

```
John 30 180.0
Alice 22 165.0
```

Static and instance methods/fields

Let's look at the code below. Here we have a class named `Human` with two fields and two methods.

```
class Human {
    String name;
    int age;

    public static void printStatic() {
        System.out.println("It's a static method");
    }

    public void printInstance() {
        System.out.println("It's an instance method");
    }
}
```

The modifier `public` isn't important for us now. It just means that other classes can also get access to our methods.

As you see, the methods `printStatic` and `printInstance` have differences in declaration.

To invoke a static method we don't need to create an object. We just call the method with the class name. In other words, you can say that a static method belongs to a class (because we don't need an object).

```
public static void main(String[] args) {
    Human.printStatic(); // will print "It's a static method"
}
```

An instance method requires a different invocation. Here we call the method `printInstance` for two different objects:

```
public static void main(String[] args) {
    Human peter =  new Human();
    peter.printInstance(); // will print "It's an instance method"

    Human alice =  new Human();
    alice.printInstance(); // will print "It's an instance method"
}
```

So, we can say that an instance method is a method that belongs to each object that we created of the particular class.

Static methods

A static method may have arguments like a regular instance method, or it may well have no arguments. But, unlike instance methods, static methods have several special features:

- a static method can access only static fields and cannot access non-static fields;
- a static method can invoke another static method, but it cannot invoke an instance method;
- a static method cannot refer to `this` keyword because there is no instance in the static context.

Instance methods, however, can access static fields and methods.

Static methods are often used as **utility methods** that are the same for the whole project. The Java library provides a lot of static methods for different classes. Here are just a few of them:

- the `Math` class has a lot of static methods, such as `Math.min(a, b)`, `Math.abs(val)`, `Math.pow(x, y)` and so on;
- the `Arrays` class has a lot of static methods for processing arrays such as `toString(...)`;
- `Long.valueOf(...)`, `Integer.parseInt(...)`, `String.valueOf(...)` are static methods too.

Here is a class with one constructor, a static method and an instance method.

```
public class SomeClass {

    public SomeClass() {
        invokeAnInstanceMethod(); // this is possible here
        invokeAStaticMethod();    // this is possible here too
    }

    public static void invokeAStaticMethod() {
        // it's impossible to invoke invokeAnInstanceMethod() here
    }

    public void invokeAnInstanceMethod() {
        invokeAStaticMethod();  // this is possible
    }
}
```

This example shows that you can invoke a static method from the instance context (constructors and instance methods), but you can't invoke an instance method from a static context.

Static fields

A **static field** is a field declared with the `static` keyword. It can have any primitive or reference type, just like a regular instance field. A static field has the same value for all instances of the class. It belongs to the class, rather than to an instance of the class.

If we want all instances of a class to share a common value, for example, a global variable, it's better to declare it as static. This can save us some memory because a single copy of a static variable is shared by all created objects.

Static variables can be accessed directly by the class name. To access a static field, you should write

```
class SomeClass {
    public static String staticStringField;
    public static int staticIntField;

    public static void main(String[] args) {
        SomeClass.staticIntField = 10;
        SomeClass.staticStringField = "it's a static member";

        System.out.println(SomeClass.staticIntField); // It prints
        System.out.println(SomeClass.staticStringField); // It print
    }
}
```

Generally, it's not a good idea to declare **non-final public static fields**, here we just used them as an example. Static fields with the `final` keyword are **class constants**, which means they cannot be changed. According to the naming convention, constant fields should always be written in uppercase with an underscore (_) to separate parts of the name.

```
class Physics {

    /**
     * The speed of light in a vacuum (m/s)
     */
    public static final long SPEED_OF_LIGHT = 299792458;

    /**
     * Electron mass (kg)
     */
    public static final double ELECTRON_MASS = 9.10938356e-31;
}
```

To use the constants, let's write the following code:

```
System.out.println(Physics.ELECTRON_MASS); // 9.10938356E-31
System.out.println(Physics.SPEED_OF_LIGHT); // 299792458
```

Instance methods

Instance methods have a great advantage: they can access fields of the particular object of the class.

```
class Human {
    String name;
    int age;

    public static void averageWorking() {
        System.out.println("An average human works 40 hours per weel
    }

    public void work() {
        System.out.println(this.name + " loves working!");
    }

    public void workTogetherWith(Human other) {
        System.out.println(this.name + " loves working with " + othe
    }

    public static void main(String[] args) {
        Human.averageWorking(); // "An average human works 40 hours

        Human peter =  new Human();
        peter.name = "Peter";
        peter.work(); // "Peter loves working!"

        Human alice =  new Human();
        alice.name = "Alice";
        alice.work(); // "Alice loves working!"

        peter.workTogetherWith(alice); // "Peter loves working with
    }
}
```

The keyword `this` represents a particular instance of the class.

Look, now we have different outputs for the method `work` because two different objects have different values for `name`. First, we created `peter` and gave him a name, then by invoking `peter.work()` we actually saw his name in the output. We did the same with `alice` and also saw her name in the output.

Look at the `workTogetherWith` method. The keyword `this` allows us to access a field of the particular object and distinguish it from the same field of another object.

Of course, instance methods can take arguments and return values just as you saw in the previous topics. Return values can be of any type including the same type as the defined class.

Constructors

Constructors are special methods that initialize a **new object** of the class. A constructor of a class is invoked when an instance is created using the keyword `new`.

A constructor is different from other methods in that:

- it has the same name as the class that contains it;
- it has no return type (not even `void`).

Constructors initialize **instances** (objects) of the class. They set values to the fields when the object is created. Also, constructors can take parameters for initializing fields by the given values.

Here is a class named `Patient`. An object of the class has a name, an age, and a height. The class has a constructor with 3 parameters to initialize objects with specific values.

```
class Patient {

    String name;
    int age;
    float height;

    public Patient(String name, int age, float height) {
        this.name = name;
        this.age = age;
        this.height = height;
    }

    public static void main(String[] args) {
        Patient patient1 = new Patient("Heinrich", 40, 182.0f);
        Patient patient2 = new Patient("Mary", 33, 171.5f);
    }
}
```

Now we have two patients, Heinrich and Mary, with the same fields, but the values of those fields are different.

To initialize the fields, the keyword `this` is used, which is a reference to the current object. Usually, the keyword `this` is used when an instance variable and a constructor or a method variable share the same name. This keyword helps to disambiguate these situations.

If you write something like `name = name`, it means that you're assigning the `name` variable to itself, which, of course, doesn't make any sense. Frankly speaking, you may distinguish two objects simply by assigning another name to the variable, like `name = newName`. It is not prohibited, but it is considered bad practice since these variables point to the same thing.

Default and no-argument constructor

The compiler automatically provides **a default no-argument constructor** for any class without constructors.

```
class Patient {
    String name;
    int age;
    float height;
}
```

We can create an instance of the class `Patient` using the no-argument default constructor:

```
Patient patient = new Patient();
```

If you define a specific constructor, the default constructor will not be created.

We can also define a constructor without any arguments, but use it to set default values for fields of a class. For example, we can initialize `name` with `"Unknown"`:

```
class Patient {

    String name;
    int age;
    float height;

    public Patient() {
        this.name = "Unknown";
    }
}
```

Such no-argument constructors are useful in cases when any default value is better than `null`.

Constructor overloading

You can define as many constructors as you need. Each constructor should have a name that matches the class name but the parameters should be different. The situation when a class contains multiple constructors is known as **constructor overloading**.

```
public class Robot {
    String name;
    String model;

    public Robot() {
        this.name = "Anonymous";
        this.model = "Unknown";
    }

    public Robot(String name, String model) {
        this.name = name;
        this.model = model;
    }
}
```

The class `Robot` has two constructors:

- `Robot()` is a no-argument constructor that initializes fields with default values;
- `Robot(String name, String model)` takes two parameters and assigns them to the corresponding fields.

To create an instance of the class `Robot` we can use either of the two constructors:

```
Robot anonymous = new Robot(); // name is "Anonymous", model is "Unk
Robot andrew = new Robot("Andrew", "NDR-114"); // name is "Andrew",
```

Bear in mind that you cannot define two constructors with the same number, types, and order of parameters!

Invoking constructors from other constructors

We can also invoke a constructor from another one. It allows you to initialize one part of an object by one constructor and another part by another constructor.

Calling a constructor inside another one is done using `this`. For example:

```
this(); // calls a no-argument constructor
```

```
this("arg1", "arg2"); // calls a constructor with two string argumer
```

Remember, the statement for invoking a constructor should be the first statement in the body of a caller constructor.

```
public class Robot {
    String name;
    String model;
    int lifetime;

    public Robot() {
        this.name = "Anonymous";
        this.model = "Unknown";
    }

    public Robot(String name, String model) {
        this(name, model, 20);
    }

    public Robot(String name, String model, int lifetime) {
        this.name = name;
        this.model = model;
        this.lifetime = lifetime;
    }
}
```

Access modifiers

An **access modifier** is a special keyword that specifies who is allowed to use your code or a special part of it. It can be placed in front of any field, method or the entire class.

Code clarity. Imagine your code is a complicated engine of a washing machine. We can cover the engine with the body and add some buttons for choosing a washing mode and starting the process. The user has no need to know what is going on inside the machine's body; the buttons to get the result are more than enough. That's how access control helps in code: you can **"hide" the engine from the user** by restricting access and simply providing them with the public "buttons".

Code safety. Now imagine you have developed a rather useful library that is used by other developers. One day some Jane Doe wants to use your code's functionality in her project, but the problem is that she needs to change one variable in one of your classes. If it is public, nothing can stop her from doing that in her code before using method A from the library. What can happen if the variable is used somewhere in method B? The B method would probably start to act unpredictably. So, protecting important parts of your code is a guarantee that it will be **used as an unmodifiable part** and its behavior will be the exact one you have developed it for.

Class visibility

A class can have one of two following modifiers:

- **package-private (default, no explicit modifier):** visible only for classes from the same package;
- **public:** visible to all classes everywhere.

The common way of using modifiers is:

- make the classes containing methods for users (the "buttons") **public**;
- make all other classes with low-level logic methods used by public ones **package-private** (cover the engine with the body).

Remember: everything that's not meant to be used/changed by classes from other packages should not be public.

Here is a class inside the package `org.java.packages.theory.p1`. with default package-private access:

```
package org.java.packages.theory.p1;

class PackagePrivateClass{

}
```

This class can be used only by other classes from the same package. It's not even visible for classes from any other package, including:

```
org
org.java.packages.theory
default package
```

Here is a public class inside the package `org.java.packages.theory.p2`

```
package org.java.packages.theory.p2;

public class PublicClass {

}
```

This class has no access restrictions, it is visible to all classes and can be used everywhere, including:

```
org
org.java.packages.theory
org.java.packages.theory.p1
default package
```

Members visibility

A class member (a field or a method, e.g. class constructor) has more options to choose from: **private**, **package-private**, **protected** and **public**. As you can see, there are two additional modifiers here. Let's consider member modifiers in more detail.

- **private** available only inside a class;
- **package-private** (also known as **default**, implicit) available for all classes in the same package;
- **protected** available for classes in the same package and for subclasses (will be covered later);
- **public** available for all classes everywhere.

Modifier	Class	Package	Subclass	All
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

The table above illustrates the level of access provided by the access modifiers: the class always has access to its members, and so on. Note that by a subclass here, we mean only a subclass of this class used in another package. We will learn about inheritance and subclasses later.

Private members

Fields are often declared **private** to control access to them from any other class.

Private methods are used to hide the internal low-level logic implementation from the rest of the code and make public methods more brief and readable.

Here is the class `Counter` with the private field `current`. This field can be read with the method `getCurrent()` , a getter method, and changed with the `inc()` method. The last one is not exactly a setter method because it doesn't manually set a value to a `current` variable, but just increments it.

```
public class Counter {
   private long current = 0;

   public long getCurrent() {
       return this.current;
   }

   public void inc() {
       inc(1L);
   }

   private void inc(long val) {
       this.current += val;
   }
}
```

Package-private members

A **package-private** access modifier does not require any keyword. If a field, a method, or a constructor has this modifier, then it can be read or changed from any class inside the same package.

The class `Salary` has a package-private field and a constructor. An instance of the `Salary` class can be created inside a `Promotion` class, and the field can also be accessed by `Promotion` and its members because they belong to the same package.

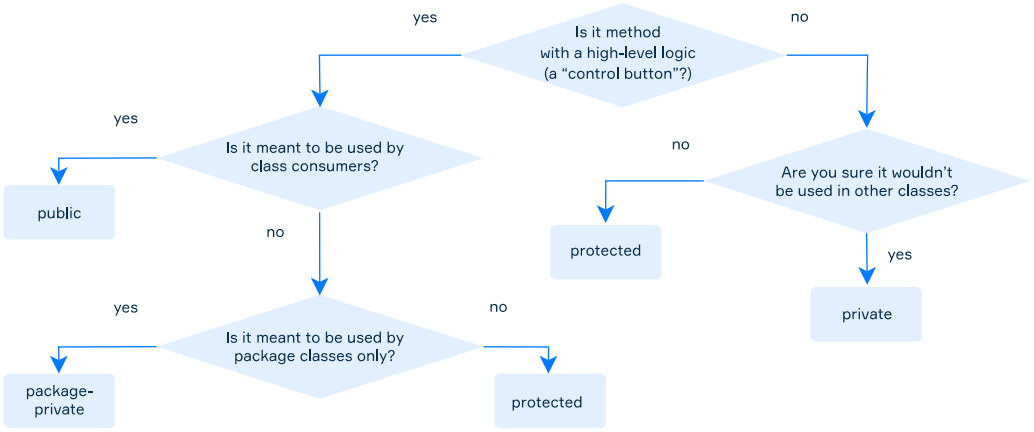
```
public class Salary {
    long income;

    Salary(long income) {
        this.income = income;
    }
}

public class Promotion {
    Salary salary;

    Promotion(Salary salary) {
        this.salary = salary;
    }

    public void promote() {
        salary.income += 1500;
    }
}
```

Protected and public members

If a class member has the **protected** access modifier, it can be accessed from classes inside the same package and all subclasses of this class (including the ones in other packages). For now, it is important to remember that the protected option is less restricting than package-private as it allows some classes from other packages access to the code member.

A **public** access modifier means that there is no restriction on using a field, method, or class. It's often used for constructors, and methods representing the class API but not commonly used with fields.

Here are common ways to understand which access modifier to choose. It is not the ultimate algorithm, because the inheritance and subclass topics have not been covered yet, but it can help you understand the main use cases of the modifiers.



Data encapsulation

In most cases, a class does not expose its fields to other classes. Instead, it makes its fields accessible through so called accessor methods.

According to the **data encapsulation** principle, the fields of a class cannot be directly accessed from other classes. The fields can be accessed only through the methods of that particular class.

To access hidden fields, programmers write special types of methods: **getters** and **setters**. Getters can only read fields, and setters can only write (modify) the fields. Both types of methods should be `public`.

Using these methods gives us some advantages:

- the fields of a class can be made read-only, write-only, or both;
- a class can have total control over what values are stored in the fields;
- users of a class don't know how the class stores its data and don't depend on the fields.

See this interesting discussion on the topic!

Getters and setters

According to the JavaBeans Convention:

- **getters** start with **get**, followed by the variable name, with the first letter of the variable name capitalized;
- **setters** start with **set**, followed by the variable name, with the first letter of the variable name capitalized.

This convention applies to any type except `boolean`. A **getter** for a boolean field starts with **is**, followed by the variable name.

The class `Account` below has four **private** fields and has **public** getters and setters for accessing them.

```
class Account {

    private long id;
    private String code;
    private long balance;
    private boolean enabled;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public long getBalance() {
        return balance;
    }

    public void setBalance(long balance) {
        this.balance = balance;
    }

    public boolean isEnabled() {
        return enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }
}
```

Let's create an instance of the class and fill the fields, then read values from the fields and output them.

```
public static void main(String[] args) {
    Account account = new Account();

    account.setId(1000);
    account.setCode("62968503812");
    account.setBalance(100_000_000);
    account.setEnabled(true);

    System.out.println(account.getId());      // 1000
    System.out.println(account.getCode());    // 62968503812
    System.out.println(account.getBalance()); // 100000000
    System.out.println(account.isEnabled());  // true
}
```

Sometimes, **getters** or **setters** can contain a more sophisticated logic. For example, **getters** may return non-stored values (calculated at runtime), or **setters** may also in some cases modify the value of another field according to changes. In the following class, the setter `setName` doesn't change the current value if the passed value is `null`.

```
class Patient {

    private String name;

    public Patient(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        if (name != null) {
            this.name = name;
        }
    }
}
```

Immutability of objects

There is an important concept in programming called immutability. **Immutable as applied to an object, means that there are no ways to change its state, i.e. none of its methods change its state, and it has no non-final protected or public fields**. In other words, immutability means that an object always stores the same values. If we need to modify these values, we should create a new object. A common example is

the standard **`String`** class. Strings are immutable objects so all string operations produce a new string. Immutable types allow you to write programs with fewer errors.

The class **`Patient`** is not immutable because it is possible to change any field of an object.

```
Patient patient = new Patient();

patient.name = "Mary";
patient.name = "Alice";
```

Records

Passing immutable data between objects is one of the most common tasks in many Java applications. Prior to Java 14, this required the creation of a class with boilerplate fields and methods, which were susceptible to trivial mistakes. Since Java 14, **records** are available to remedy these problems.

Commonly, we write classes to simply hold data, such as database results, query results, or information from a service. In many cases, this data is immutable, since immutability ensures the validity of the data.

To accomplish this, we create data classes with the following:

- private, final field for each piece of data
- getter for each field
- public constructor with a corresponding argument for each field
- equals method that returns true for objects of the same class when all fields match
- hashCode method that returns the same value when all fields match
- toString method that includes the name of the class and the name of each field and its corresponding value

For example, we can create a simple Person data class with a name and address:

```
public class Person {

    private final String name;
    private final String address;

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

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (!(obj instanceof Person)) {
            return false;
        } else {
            Person other = (Person) obj;
            return Objects.equals(name, other.name)
              && Objects.equals(address, other.address);
        }
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", address=" + address + "]"
    }

    // standard getters
}
```

While this accomplishes our goal, there are two problems with it:

- **There's a lot of boilerplate code**: we have to repeat the same tedious process for each data class, creating equals, hashCode, and toString methods; and creating a constructor that accepts each field. While IDEs can automatically generate many of these classes, they fail to automatically update our classes when we add a new field.
- **We obscure the purpose of our class**: the extra code obscures that our class is simply a data class that has two String fields, name and address.

Records are immutable data classes that require only the type and name of fields. The equals, hashCode, and toString methods, as well as the private, final fields

and public constructor, are generated by the Java compiler. To create a Person record, we can simply use:

```
public record Person (String name, String address) {}
```

Enums

Java 5, first introduced the **enum** keyword. It denotes a special type of class that always extends the java.lang.Enum class. Constants defined this way make the code more readable, allow for compile-time checking, document the list of accepted values upfront, and avoid unexpected behavior due to invalid values being passed in.

Here's a quick and simple example of an enum that defines the status of a pizza order; the order status can be NOT_ASSIGNED, ORDERED, READY or DELIVERED:

```
public class Pizza {
    private PizzaStatus status = PizzaStatus.NOT_ASSIGNED;

    public enum PizzaStatus {
        NOT_ASSIGNED,
        ORDERED,
        READY,
        DELIVERED
    }

    public boolean isDeliverable() {
        if (getStatus() == PizzaStatus.READY) {
            return true;
        }
        return false;
    }

    // Methods that set and get the status variable.
}
```

A basic enum type in Java does not contain any public fields, nor any methods that change state. So in this case it would be immutable. However, you can add fields and methods to an enum type. If you add methods that allow you to set fields, for example, then that enum type would be mutable. See the *Resources* Section at the end for more details.

Sharing references

More than one variable can refer to the same object. It is important to understand that two variables refer to the same data in memory rather than two independent copies. Since our class is mutable, we can modify the object using both references.

```
public static void main(String[] args) {
    Patient patient = new Patient();

    patient.name = "Mary";
    patient.age = 24;

    System.out.println(patient.name + " " + patient.age); // Mary 24

    Patient p = patient;

    System.out.println(p.name + " " + p.age); // Mary 24
}
```

Operations on references

Only the relational operators == and != are defined for object reference:

- The equality condition tells you whether two references point to the same object in memory
- The equality condition is evaluated on the values of the references (i.e., addresses in memory)!

As for any reference type, a variable of a class type can be **null** which means it is not initialized yet.

```
Patient patient = null;
```

Garbage collector

Given the name, it seems like *Garbage Collection* would deal with finding and deleting the garbage from the memory. However, in reality, *Garbage Collection* tracks each and every object available in the JVM heap space, and removes the unused ones.

Basically, *GC* works in two simple steps, known as Mark and Sweep:

- **Mark** this is where the garbage collector identifies which pieces of memory are in use and which aren't.
- **Sweep** this step removes objects identified during the "mark" phase.

Advantages:

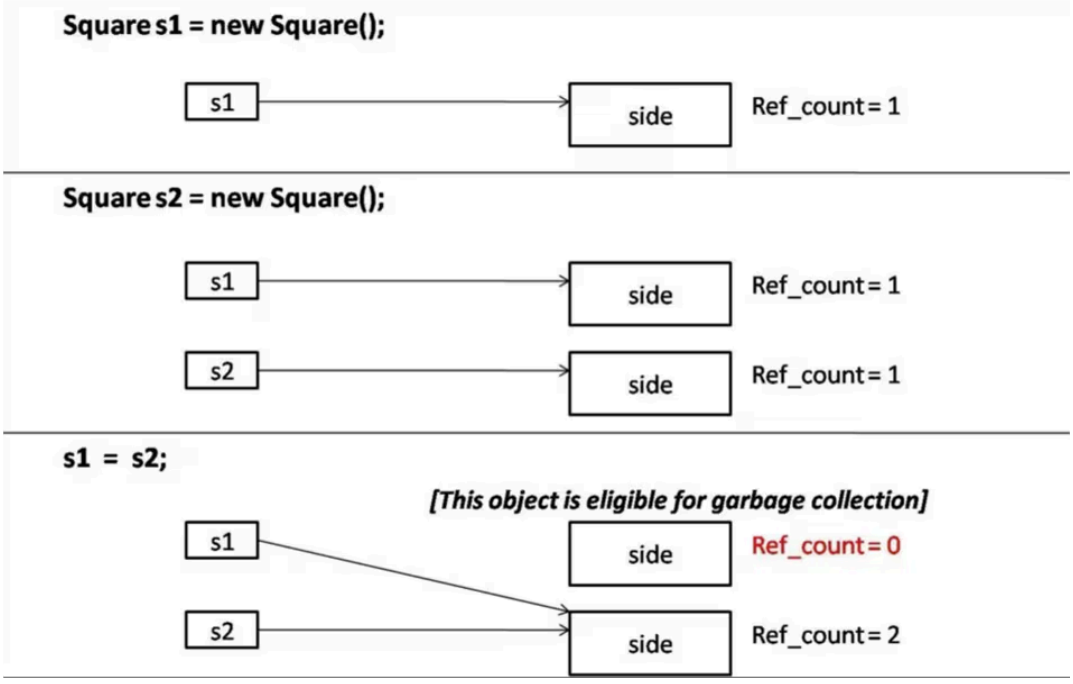
- No manual memory allocation/deallocation handling because unused memory space is automatically handled by *GC*
- No overhead of handling *Dangling Pointer*
- Automatic *Memory Leak* management (*GC* on its own can't guarantee the full proof solution to memory leaking; however, it takes care of a good portion of it)

Disadvantages:

- Since *JVM* has to keep track of object reference creation/deletion, this activity requires more CPU power than the original application. It may affect the performance of requests which require large memory.
- Programmers have no control over the scheduling of CPU time dedicated to freeing objects that are no longer needed.
- Using some GC implementations might result in the application stopping unpredictably.
- Automatized memory management won't be as efficient as the proper manual memory allocation/deallocation.

GC Implementations

- Serial: the single-threaded GC, a simple but efficient collector for small memory footprint
- Parallel: the multithreaded GC, ideal for throughput-oriented applications
- G1: the multithreaded GC, designed to offer a good tradeoff among latency and throughput
- Z and Shenandoah: the new low-latency GCs available from OpenJDK 15



Wrapper classes

Each primitive type has a class dedicated to it. These classes are known as **wrappers**, and they are **immutable** (just like strings). Wrapper classes can be used in different situations:

- when a variable can be `null` (absence of a value);
- when you want to use special methods of these classes (e.g., for converting to and from strings).
- when you need to store values in generic collections (will be considered in the next topics);

The following table lists all primitive types and the corresponding wrapper classes. As you can see, Java provides eight wrapper classes: one for each primitive type.

Primitive	Wrapper
boolean	Boolean
byte	Byte
char	Character
int	Integer
float	Float
double	Double
long	Long
short	Short

Boxing and unboxing

Boxing is the conversion of primitive types to objects of corresponding wrapper classes. **Unboxing** is the reverse process. The following code illustrates both processes:

```
int primitive = 100;
Integer reference = Integer.valueOf(primitive); // boxing
int anotherPrimitive = reference.intValue();    // unboxing
```

Auto-boxing and **auto-unboxing** are automatic conversions performed by the Java compiler.

```
double primitiveDouble = 10.8;
Double wrapperDouble = primitiveDouble;        // autoboxing
double anotherPrimitiveDouble = wrapperDouble; // auto-unboxing
```

Converting primitive types to and from strings

Wrapper classes can also be used for converting primitive types into strings and vice-versa.

```
String s = Integer.valueOf(34).toString();
int n = Integer.valueOf("34").intValue();
```

Comparing wrappers

Just like for any reference type, the operator == checks whether two wrapper objects are actually equal, i.e. if they refer to the same object in memory. The method `equals`, on the other hand, checks whether two wrapper objects are meaningfully equal, for example, it checks if two wrappers or strings have the same value.

```
Long i1 = Long.valueOf("2000");
Long i2 = Long.valueOf("2000");
System.out.println(i1 == i2);      // false
System.out.println(i1.equals(i2)); // true
```

Do not forget about this feature when working with wrappers. Even though they correspond to primitive types, wrapper objects are reference types!

NPE when unboxing

There is one possible problem when unboxing. If the wrapper object is `null`, the unboxing throws a`NullPointerException`.

```
Long longVal = null;
long primitiveLong = longVal; // It throws an NPE
```

Another example is arithmetic operations on numeric wrapper types. They may cause an **NPE** since unboxing is involved.

```
Integer n1 = 50;
Integer n2 = null;
Integer result = n1 / n2; // It throws an NPE
```

Resources

- https://www.baeldung.com/jvm-garbage-collectors
- https://www.baeldung.com/java-record-keyword

- https://www.baeldung.com/a-guide-to-java-enums