# Java – Threads and intrinsic locks

## 1. Java and OOP background fundamentals

### 1.1. Objects, methods and data

One significant advantage of OOP (object oriented programming) is data encapsulation. Each object holds its own data (called fields, attributes or variables, they can be objects themselves). Each object also defines a set of functions (called **methods**) to externally (by other objects) interact with the object.

Both data and methods can be public or private, **public** means other objects can access and use them, **private** means only the object itself can access them. All object's data should be private and accessed only indirectly through a set of public methods. This is what encapsulation is about, the set of public methods of an object is the **object's interface**.

By making object's data private and enforcing all external accesses to the object to be made through its interface (public methods) a data abstraction layer is created. An object can be completely internally redesigned, as far as the interface methods keep working the same way, no other objects will have to be modified.

In Java, object´s methods and data are identified by names, they should start with a lowercase letter (Java naming conventions). Often when a method is called it needs caller provided data, for that purpose, when a method is defined the arguments the caller should use are also defined (they may be none), also a method can return a result to the caller, this is also established when the method is defined.

Java and OOP in general support **polymorphism**, for instance an object can have several methods with the same name, but this is allowed only if each defined method with the same name has different arguments. The number of arguments may be the same as far as they are of different types. Although having the same name they are in fact distinct methods. The correct method to be used is settle accordingly from the arguments the caller is using. The method´s name together with its argument types is called the method's **signature**, and that must be unique within each object.

### 1.2. Classes and constructors

An object is created from and object definition, called a **class**. A class defines the object properties, namely its data and methods, in Java a class it identified by a class name, and it should start with an uppercase letter (Java naming conventions).

Once a class is defined objects can be created from it, they are called class instances or simply **instances**. Each instance of a class is an independent object with its own data and methods copied from the class definition.

When an object is created (class instantiation) it may be suitable to initialize some of its data, to achieve that every class has a special public method called **constructor**, it has the same name of the class and returns an instance of the class (object). When declaring a constructor, the returned type cannot be defined because it's implicitly the class type.

By default, every class has one constructor with no arguments, and it does nothing. This is the default constructor if none is declared in the class. Also several constructors are possible (polymorphism), as far as they have different arguments. Constructors are typically used to perform initialization operations required to obtain a ready to use object.

In Java an object is created by using the **new** keyword followed by a call to a class constructor method (whose name matches the class name).

## 1.3. Classes and primitive types

A software program performs operations on data (called fields, attributes or variables), different data types must be used for different purposes. In a pure OOP language, the data type concept is entirely replaced by the class concept.

Nevertheless, in Java, for the sake of efficiency, the so called primitive types also exist, so a variable type can be either a class or a primitive type. Some Java primitive types are integers (byte, short, int, long), in Java, all integer primitive types are signed. Other Java primitive types are float, double, boolean, and char.

A primitive type variable stores a value accordingly to its type. When a variable is declared, it´s initial value is also settled, for numbers, the zero value is assigned, for boolean, false.

A non-primitive type variable is pointer to an object, so the variable type should be the pointed object's class. In this case we say points or refers and not stores like with primitive types. Non-primitive type variables are initially assigned the **null** value, this means it does no point to any object.

There is a major difference between assigning a primitive type variable or a non-primitive type variable. Assigning a primitive type variable means copying the assigned value and store it in the variable. Assigning a non-primitive type means just changing the object to where it's pointing to.

This is a difference that must be kept in mind, imagine instance the implementation of a method to increment one unit to an integer:

```
public void inc(int value) { value=value+1;}
```

This will not work. Because the argument is a primitive type, the caller provided value is copied to argument value, and thus the increment is done over the copy and not over the original.

Also be aware that, again for the sake of efficiency, Java compilers perform autoboxing and unboxing, for primitive types. It's an automatic conversion between primitive types and equivalent object classes during assignments. Consequently, the above sample wouldn't work either if the Integer class was used as argument type because it would be converted to int on assignments and vice versa.

## 1.4. Static and non-static

It was mentioned before that an object is created by instantiating a class, and that consists of copying all the class methods and data (variables) to the new object, well, that is not entirely true.

In Java, by default, class methods and data are **non-static**, and those are in fact copied to the object when created. From that instant on, they are independent and have no relation whatsoever with the class.

Yet, in a class, other methods and data can be declared as **static**, they are not copied to the object, they exist only in the class. So what are they useful for?

The primary purpose of a class is being used to create objects, but that includes only the non-static elements of the class. The static elements of a class have another use. In Java and other OOP languages a class also works itself as an object, it can be called the **class object**. For each class **there is only one instance of the class object** and

it's **identified by the class name**. The class object simply exists with no instantiation. Because it contains only the static elements of the class, static methods can use only other static elements (methods and variables).

The class object is single and global, its **public elements** can be used by any object simply by referring the class name.

In a class instance (object created by the class instantiation) there are only non-static class elements, however, static elements are also accessible, nevertheless they don't belong to the object, they belong to the class object, so they are globally unique. Because the object belongs to the class, within the object, class static elements can be referred directly without specifying the class name, also **private static** elements of the class will be accessible. Static elements of other classes are accessible only if they are public.

This direct reference to the class static elements within a class instance can lead to some confusion, it must be remembered these elements are stored in the class object and not it the class instance (object).

Static elements also solve an OOP issue, in order for an application to start running an initial object is required. Once an initial object exists, it can then can create other objects through class instantiation. Because the static elements of classes exist initially they can be used to solve this issue.

It Java this initial object is usually a class object that defines the **main()** static method, because it's static it simply exists, therefore it can be used as the application run starting point.

## 1.5. Inheritance

It's an OOP central concept, a class can be defined by reusing an existing class definition. One way to achieve this in Java is by declaring the new class (subclass) **extends** an existing class (superclass). By doing so, the subclass inherits all public methods and data from the original superclass, new methods and data can then be defined in the subclass, they are added to inherited methods and data.

Even though private elements of the superclass are not directly available in the subclass, if the superclass has public methods that access private elements, those methods can be used in the subclass and will have access to those private elements.

Methods inherited from the superclass can be redefined, if a method definition has the same signature (name and arguments) and the same return type as in the superclass, the new implementation replaces (overrides) the one defined in the superclass. The **@override** annotation should be used in this cases, it will generate a compile error if the method is not defined in the superclass.

## 1.6. Interfaces

An interface definition is similar to a class definition, as with classes they should be identified by a name starting with an uppercase letter. Like a class, an interface defines methods and data, but unlike a class, it cannot be instantiated to create objects.

The purpose of defining an interface is meeting the previously described object's interface concept, it establishes public characteristics, but doesn't actually describe how they are implemented. An interface establishes a standard, later, a class definition can declare it implement an interface, and thus, it's declaring it makes available the features described it that interface declaration.

All methods defined in an interface are implicitly **public**, they can be either **abstract**, **default** or **static**. By default they are **abstract**.

An abstract method has no implementation, just a return type and a signature (name and arguments), it has no body. Abstract method's implementation is left to do for classes that implement the interface. Default and static methods in an interface have an implementation.

An interface's variables are constants, they are implicitly **public**, **static**, and **final**. Final means the value is settled on the declaration and can't be changed subsequently.

As with classes, inheritance can be used. However, an interface can extend one or a comma separated list of existing interfaces, a class can only extend one class.

### 1.7. Multiple Inheritance

Multiple inheritance means, of course, inheriting definitions from several origins. As just seen, regarding interfaces, this is possible through declaration **extends**.

Though a class can extend only one superclass, in addition, it can declare it implements one or several interfaces, and this is also a form of inheritance.

One issue arising from multiple inheritance is method's signatures collision, several precedence rules apply for these cases depending on the methods being static or abstract. For instance non-static methods take precedence over interface default methods. Of course, redefining a superclass static method as non-static in a subclass is illegal.

### 1.8. Packages in Java

A package is a group of related type definition (classes, interfaces and others), type definitions within a package are package members. A package is identified by a name, it should be a low case letters name to avoid conflicts with classes and interfaces.

A package is an independent naming space, for instance two classes with the same name can be defined, as far as they belong to different packages (members of different packages). Type definitions in a package are available to all that package members, a member of a package can use other packages type definitions by explicitly importing them, however, only public type definitions can be imported. By default classes and interfaces are private, this means they can be used only within the package. To make a type definition accessible to other packages, they must be explicitly declared as public.

## 2. Multi-thread programming in Java

### 2.1. Threads

A thread is an independent execution flow, when a Java program starts running it will have only one thread that usually executes the body of the main() static method defined in some class. Though, from this initial single thread, other threads can be started without stopping the initial thread, all started threads and the initial thread run in parallel at the same time.

One key issue arising from multi-threading is concurrency, all threads are running at the same time and they all have access to the same data. If two or more threads access the same piece of data at the same time, results become unpredictable and may even lead to an application deadlock. For safe multi-thread programming, concurrency control mechanisms must be enforced.

Two approaches can be used for threads in Java, the High Level Concurrency Objects provides **Executor** interfaces like **ExecutorService** or **ScheduledExecutorService**, a lower level approach to be used here is directly managing the threads.

## 2.2. Defining a thread starting point

In Java a thread starting point (where the thread starts running as an independent task) is an object that implements the Runnable interface, this interface defines the **run() abstract method**, this method's body is to be implemented and it's the thread's starting point.

To instantiate such an object, a class that implements the Runnable interface must be defined in the first place. This can be either a class that extends the Thread class (which in turn implements the Runnable interface) or a class that directly implements the Runnable interface. In either case the run() method's body must be defined:

```
public class MyThreadClass1 extends Thread {
   @Override
   public void run() {          // thread execution starting point
   // code to be executed
   }                            // thread execution ends
}
```

```
public class MyThreadClass2 implements Runnable {
   public void run() {          // thread execution starting point
   // code to be executed
   }                            // thread execution ends
}
```

## 2.3. Starting a thread

The thread itself is managed by an object of the Thread class (or a subclass), among others, the Thread class defines the start() method, this method creates the thread and calls the run() method of the Runnable interface.

Depending on how the class that implements the run() method was defined, the thread will be started in slightly different way.

If the class extends the Thread class (MyThreadClass1), because it's a Thread's subclass, it can be used to start and control the thread:

```
MyThreadClass1 myThread = new MyThreadClass1();
myThread.start();
```

If the class just implements the Runnable interface (MyThreadClass2), then a Thread instance must also be created, one of the Thread class constructors receives a runnable interface as argument:

```
MyThreadClass2 myThreadObj = new MyThreadClass2();
Thread myThread = new Thread(myThreadObj);
myThread.start();
```

In either case, when the start() method is called the thread starts running in background and calls the run() method.

The Thread class provides several other methods for thread control, including the sleep() static method to pause the current thread execution during a time period,

interrupt() to interrupt a thread's execution and join() to wait for a thread´s to end its execution.

## 3. Concurrency control

In multi-threading several independent threads are running simultaneously and they may access the same objects and methods, concurrent access means accessing the same object or method simultaneously, this leads to unpredictable results and must be avoided.

Java provides a number of ways to control concurrent access, including semaphores that can be used to limit the number of threads accessing something.

Here the focus will be on a very simple mechanism called **lock**, also known as a mutex (from mutual exclusion). A lock has two states: **acquired** (owned by a thread) or **released**. If acquired by one thread it cannot be acquired by another thread until it's released by the first. Even if two threads try to acquire a lock at the exact same time, it's guaranteed only one will have success, the other will be **blocked and waiting** until the first releases the lock.

### 3.1. Java intrinsic locks

In Java, **every object and every class has an associated mutual exclusion lock dedicated to it**. These intrinsic locks are used through the **synchronized** statement.

When a **non-static method** is declared to be synchronized, the **object's intrinsic lock** is acquired by the calling thread when the method is called and released when the method returns.

When a **static method** is declared to be synchronized, the **class's intrinsic lock** is acquired by the calling thread when the method is called and released when the method returns.

Thus, calling a synchronized static method and a synchronized non-static method of the same class acquire different locks, so it will not enforce mutual exclusion.

The following class implementation if faulty because it does not take this in account:

```
public class Counter {
   private static int value;
   public synchronized int get() { return value; }
   public static synchronized void inc() { value++; }
   public static synchronized void dec() { value--; }
   public static synchronized void reset() { value=0; }
}
```

Even though all methods are declared to be synchronized, the get() method is non-static, whereas other methods are static, thus mutual exclusion is not enforced between the get() method execution and other methods execution.

Declaring synchronized methods guarantees only one thread will be ever executing that method at any time. Because the used intrinsic lock is associated with the object (for non-static methods) or the class (for static methods) it also enforces mutual exclusion with other synchronized static methods of the class, for the first case, and other synchronized non-static methods of the object for the second case.

### 3.2. Synchronized blocks

Using the synchronized statement in methods declaration in very simple and straightforward, however, in many cases, it may be a too coarse-grained locking.

Imagine a new method capable of reading a value from the command line and storing it in the class is intended, by following the previous approach, it would be something like this:

```
public class Counter {
   private static int value;
   public static synchronized int get() { return value; }
   public static synchronized void inc() { value++; }
   public static synchronized void dec() { value--; }
   public static synchronized void reset() { value=0; }
   public static synchronized void readVal() throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Please enter a new value: ");
      int v = Integer.parseInt(in.readLine());
      value=v;
   }
}
```

The issue with this implementation is that during the whole execution of the readVal() method the class's intrinsic lock is acquired, so other threads are banned from calling any method of the class. But in fact the only execution requiring an exclusive access is the method's last line. Definitely, in this sample, the time during which the whole class is locked can be rather high.

As general principle, an effort should be made to minimize the time during which any lock is acquired, this favours performance as it becomes less likely to have other threads waiting.

We could solve this issue by creating a new synchronized method, just for the assignment operation, then the readVal() method wouldn't need to be synchronized any more:

```
public class Counter {
   private static int value;
   public static synchronized int get() { return value; }
   public static synchronized void inc() { value++; }
   public static synchronized void dec() { value--; }
   public static synchronized void reset() { value=0; }
   public static synchronized void setVal(int v) { value=v; }
   public static void readVal() throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Please enter a new value: ");
      int v = Integer.parseInt(in.readLine());
      setVal(v);
   }
}
```

Even though defining synchronized methods for each basic operation requiring mutual exclusion is not a bad idea, a finer intrinsic lock control can also be achieved by using the synchronized statement in another form.

It's possible to define an arbitrary block of code to have its execution synchronized with any object or class, provided as argument to the synchronized statement. The result is that, before the block of code execution starts, the referenced object or class intrinsic lock is acquired, and then released when that block´s execution ends.

To integrate this use of the synchronized statement with synchronized methods declaration, the referenced object must be the same. To lock the class the class should be referenced in the form **{ClassName}.class**, to lock the object, the object´s name. Within non-static methods, **this** can be to reference the own object where the method is

being executed. Also, within an instance (non-static method), the object´s class can be obtained by calling **this.getClass()**.

So another solution for the previous issue is making the assignment within a block synchronized with the class:

```
public class Counter {
   private static int value;
   public static synchronized int get() { return value; }
   public static synchronized void inc() { value++; }
   public static synchronized void dec() { value--; }
   public static synchronized void reset() { value=0; }
   public static void readVal() throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Please enter a new value: ");
      int v = Integer.parseInt(in.readLine());
      synchronized(Counter.class) { value=v; }
   }
}
```

Another case for the use of synchronized blocks is when and object or class contains independent parts and each part is accessed by a distinct set of methods, there is no need for methods accessing one part to block methods accessing the other part.

The following nonsense example tries to illustrate this:

```
public class CounterAB {
   private static int valueA;
   private static int valueB;
   private Object lockA = new Object();
   private Object lockB = new Object();
   public static int getA() { synchronized(lockA) { return valueA; } }
   public static int getB() { synchronized(lockB) { return valueB; } }
   public static void incA() { synchronized(lockA) { valueA++; } }
   public static void incB() { synchronized(lockB) { valueB++; } }
   public static void decA() { synchronized(lockA) { valueA--; } }
   public static void decB() { synchronized(lockB) { valueB--; } }
   public static void resetA() { synchronized(lockA) { valueA=0; } }
   public static void resetB() { synchronized(lockB) { valueB=0; } }
}
```

Methods handling valueA acquire lockA, and methods handling valueB acquire lockB, so they don’t interfere with each other’s execution. Because primitive types don’t have intrinsic locks, alternative objects are created just for the purpose of locking accesses to each variable.

---

References: Java Documentation - https://docs.oracle.com/en/java/