# Smart contract Java

## Prerequisites

To use the Java Kalima API, it is recommended to have previously read the documentation API_Kalima and to have installed the Java JDK in version 11 at least.

A complete Java smart contract example is available on our public GitHub: KalimaJavaExample.

To create a Java smart contract, you just need the **Kalima.jar** API, so you should implement this API in your project to be able to create your Java smart contract

## 1-Smart contracts?

A Smart Contract is a computer program that automates the execution of a set of predefined instructions when prerequisites are met.

During its execution, all validation steps are recorded at the blockchain level. These records ensure the security of all data by preventing its modification or deletion.

**Smarts Java contracts == "Memcachecallback" which we will see later**

**Java smart contracts** are in the form of a java class that implements the callback named MemCacheCallback. This callback is implemented to react to certain events such as the creation of a new cache or the arrival of new transactions.

### MemCacheCallback

You can add one MemCache callback per address to react to the arrival of new transactions.

The MemCacheCallback interface contains the following callbacks:

- getAddress: Must return the corresponding address
- putData: This callback is called when new cached data arrives (adding, updating a current value)
- removeData: This callback is called when a current value is deleted

### Example

For example, we have the following addresses:

- /sensors: Will contain sensor values, such as temperatures for example
- /alarms/fire: Will contain fire alarms

We want a silk fire alarm to be created when a temperature exceeds 100°C.

We also want to display in the console the arrival of new temperatures as well as the arrival of fire alarms.

When we restart our node, we do not want the data already present to be processed again, to avoid duplicate alarms.

We start by implementing two different MemCacheCallbacks (one for the /sensors address, one for the /alarms/fire address):

```java
public class SensorsCallBack implements MemCacheCallback{

	private String address;
	private Clone clone;

	public SensorsCallBack(String address, Clone clone) {
		this.address = address;
		this.clone = clone;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg msg = KMsg.setMessage(kMessage);
		String key = msg.getKey();
		System.out.println("new sensor value key=" + key + " body=" +
new String(msg.getBody()));
		if(key.equals("temperature")) {
			int temperature = Integer.parseInt(new
String(msg.getBody()));
			if(temperature >= 100) {
				clone.put("/alarms/fire", "temperature",
("Temperature too high: " + temperature + " °C").getBytes());
			}
		}
	}

	@Override
	public void removeData(KMessage kMessage) {

	}

	@Override
	public String getAddress() {
		return address;
	}
}
```

```java
public class AlarmsCallback implements MemCacheCallback {

	private String address;

	public AlarmsCallback(String address) {
		this.address = address;
	}

	@Override
	public String getAddress() {
		return address;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg kMsg = KMsg.setMessage(kMessage);
		System.out.println("new alarm key=" + kMsg.getKey() + " body="
+ new String(kMsg.getBody()));
	}

	@Override
	public void removeData(KMessage kMessage) {

	}
}
```

As we can see, our two implementations are relatively simple. SensorsCallback simply displays the body of the messages received, then, if the key of the data received is "temperature", it will control the temperature, and create a new transaction at the address /alarms/fire if the temperature is greater than or equal to 100 ° C.

AlarmsCallback simply displays the body of the received data.