DATA SERIALIZATION
Serialization is used when:

- Transferring data over the network
- Spilling data to disk
- Caching to memory serialized
- Broadcasting variables
Java serialization vs. Kryo serialization

Java serialization:
- Uses Java’s `ObjectOutputStream` framework
- Works with any class you create that implements `java.io.Serializable`
- You can control the performance of serialization more closely by extending `java.io.Externalizable`
- Flexible, but quite slow
- Leads to large serialized formats for many classes

Kryo serialization:
- Recommended serialization for production apps
- Use Kryo version 2 for speedy serialization (10x) and more compactness
- Does not support all `Serializable` types
- Requires you to register the classes you’ll use in advance
- If set, will be used for serializing shuffle data between nodes and also serializing RDDs to disk

```java
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```
To register your own custom classes with Kryo, use the `registerKryoClasses` method:

```scala
val conf = new SparkConf().setMaster(...).setAppName("")
conf.registerKryoClasses(Seq(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

- If your objects are large, you may need to increase `spark.kryoserializer.buffer.mb` config property

- The default is 2, but this value needs to be large enough to hold the largest object you will serialize.
TUNING FOR Spark

High churn

Low churn
Cost of GC is proportional to the # of Java objects

(so use an array of Ints instead of a LinkedList)

High churn

To measure GC impact:

-verbose:gc -XX:+PrintGCDetails   -XX:+PrintGCTimeStamps
<table>
<thead>
<tr>
<th></th>
<th>Parallel GC</th>
<th>Parallel Old GC</th>
<th>CMS GC</th>
<th>G1 GC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-XX:+UseParallelGC</td>
<td>-XX:+UseParallelOldGC</td>
<td>-XX:+UseConcMarkSweepGC</td>
<td>-XX:+UseG1GC</td>
</tr>
<tr>
<td>-XX:ParallelGCThreads=&lt;#&gt;</td>
<td>-XX:ParallelOldGCThreads=&lt;#&gt;</td>
<td>-XX:ParallelCMSThreads=&lt;#&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uses multiple threads to do young gen GC</td>
<td>Uses multiple threads to do both young gen and old gen GC</td>
<td>Concurrent Mark Sweep aka “Concurrent low pause collector”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Will default to Serial on single core machines</td>
<td>Also a multithreading compacting collector</td>
<td>Tries to minimize pauses due to GC by doing most of the work concurrently with application threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aka “throughput collector”</td>
<td>HotSpot does compaction only in old gen</td>
<td>Uses same algorithm on young gen as parallel collector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Good for when a lot of work is needed and long pauses are acceptable</td>
<td></td>
<td>Use cases: Garbage First is available starting Java 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use cases: batch processing</td>
<td></td>
<td>Designed to be long term replacement for CMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is a parallel, concurrent and incrementally compacting low-pause GC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>