Core Java Programming

Presented By:

Corporate Trainer

Mallikarjuna G D

gdmallikarjuna@gmail.com

CONTENTS

- ✓ EXCEPTION HANDLING
- ✓ MULTITHREADING
- ✓ IO STREAMS
- ✓ COLLECTION FRAMEWORK
- ✓ GENERICS
- ✓ ANNOTATIONS

Exception Handling:

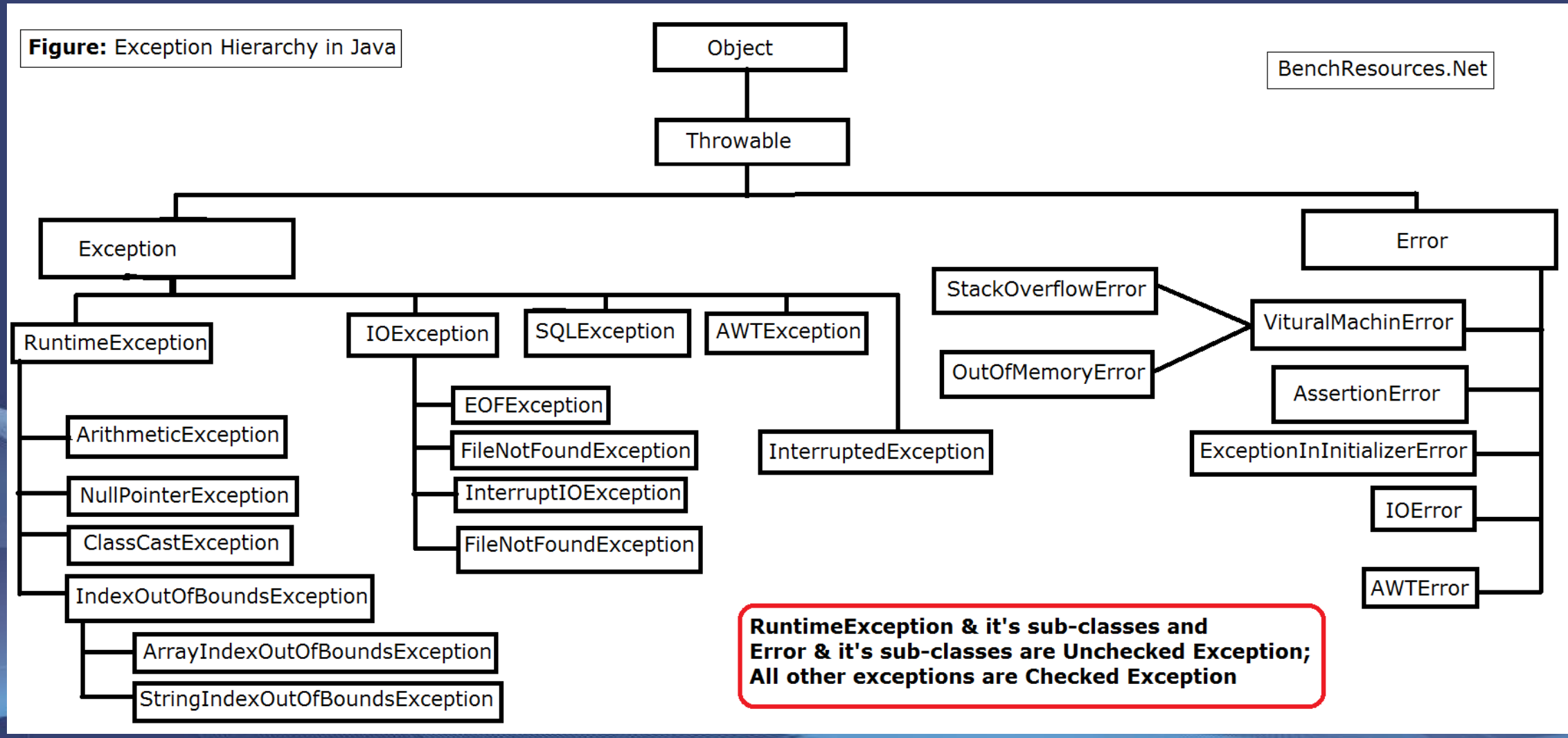
- Exception is an event that interrupts the normal flow of the program that results to abnormal termination of program .
- In java exceptional handling is a mechanism to handle the runtime errors so that normal flow of the application can be maintained.
- Maintains the normal flow of the application.
- Gives alternate way to normal flow. Or Graceful termination to the program.
- Separates the normal code & risky code.

Example:

```
statement 1;
statement 2;
statement 3;  //exception occurs
statement 4;
statement 5;
```

There are 5 statements in program and there occurs an exception at statement 3, rest of the code will not be executed i.e. statement 4 & 5 will not run. If we perform exception handling, rest of the statement will be executed..

Exception Hierarchy:



Types of Exception :

The sun micro system says there are three types of exceptions: Checked, Unchecked and Error.

There are mainly two types of exceptions: where error is considered as unchecked exception.

Checked Exception	Unchecked Exception
Subclass of Exception	Subclass of RunTimeException & Errors
Checked by compiler whether you are handling or not.	Not checked by compiler whether you are handling or not.
Force to developer to handle by either writing try/catch or throws.	Do not force to developer to handle.
Also called as Caught Exceptions	Also called as Uncaught Exceptions

Keywords:

- try { }

  try block is used to enclose the code that might throw an exception.
  It must be used within the method.

- catch { }

  catch block is used to statement which will catch the arisen exception.

- finally { }

  It is a block that is used to execute important code without fail like closing connection.

Keywords (Continue..):

- throw:

  It is used to explicitly throw an exception. throw either checked or unchecked exception in java.

  Syntax: throw new IOException("sorry device error);

- throws:

  It is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

  Syntax:

  return_type method_name() throws exception_class_name{ //method code}

Example for try, catch & finally keywords :

```
class TestBlocks{
  public static void main(String args[]){
    try{
        int data=25/0;
       System.out.println(data);
        }
      catch(ArithmeticException e) {
       System.out.println(e); }
    finally {
            System.out.println("finally block executes always"); }
        System.out.println("rest of the code...");
 }
 }   /* java.lang.ArithmeticException: / by zero
        finally block  executes always
        rest of the code */
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Example for throw keyword:

```
public class TestThrow {
   static void validate(int age) {
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]) {
      validate(13);
      System.out.println("rest of the code...");
  }
} /* O/P:
Exception in thread main java.lang.ArithmeticException:not valid */
```

Difference b/w throw & throws:

throw	throws
It is used to explicitly throw an exception.	It is is used to declare an exception
Checked exception cannot be propagated using throw only	Checked exception can be propagated with throws
Throw is followed by an instance	Throws is followed by class.
Throw is used within the method.	Throws is used along with method signature.
You cannot throw multiple exceptions	You can declare multiple exceptions public void method()throws IOException, SQLException

CUSTOM EXCEPTION

It provides the flexibility to add your attributes and methods that are not part of the standard exception. This helps to throw messages which are application-specific error codes and messages.

Benefits

- It shares additional information or functionality which is not of Java Standard exception (it can't provide any additional benefits using UnsupportedOperationException or IllegalArgumentException)
- Follow naming convention similar to other standard exceptions ending with Exception (similar to ArithmaticException, MyCustomException so on)
- Provide good documentation
  - /**
  - My business exception
  - Some error message
  - @author Mallikarjuna G D */

  Public class MyCustomException extends Exception{ … }

CUSTOM EXCEPTION

- Provide the constructor with gives the cause
  - public void method(String data) throws MyCustomException{

```
try{


}catch(NumberFormatException e){

 throw new MyCustomException("my error det",e,ErrorCode.INVALID_INPUT);
}

}
```

```
public class MyCustomException extends Exception{
   public MyCustomException(String message, Throwable cause, ErrorCode
  code){
    super(message, cause);
    this.code = code;
  }
}
```

CUSTOM EXCEPTION

```
/**
 * The MyCustomException wraps all checked standard Java exception and enriches them with a custom error code.
 * You can use this code to retrieve localized error messages and to link to our online documentation.
 *
 * @author Mallikarjuna G D
 */
public class MyCustomException extends Exception {

		private static final long serialVersionUID = 234556777788L;

		private final ErrorCode code;

		public MyCustomException(ErrorCode code) {
			super();
			this.code = code;
		}

		public MyCustomException(String message, Throwable cause, ErrorCode code) {
			super(message, cause);
			this.code = code;
		}

		public MyCustomException(String message, ErrorCode code) {
			super(message);
			this.code = code;
		}

		public MyCustomException(Throwable cause, ErrorCode code) {
			super(cause);
			this.code = code;
		}

		public ErrorCode getCode() {
			return this.code;
		}
}
```

CUSTOM EXCEPTION

```
public void method() {
		try {
			myException(new String("99999999"));
		} catch (MyCustomException e) {
			// handle exception
			log.error(e);
		}
}

private void myException(String input) throws MyCustomException {
		try {
			// do something
		} catch (NumberFormatException e) {
			throw new MyCustomException("A message that describes the error.", e, ErrorCode.INVALID_INPUT);
		}
}
```

CUSTOM EXCEPTION

```
public void method() {
        try {
                myException(new String("99999999"));
        } catch (MyCustomException e) {
                // handle exception
                log.error(e);
        }
}

//no need throws
private void method(String input) {
        try {
                // do something
        } catch (NumberFormatException e) {
                throw new MyUncheckedCustomException("A message that describes the error.", e,
        ErrorCode.INVALID_INPUT);
        }
}
```

```
        public MyUnCheckedCustomException(String message, ErrorCode code) {
                super(message);
                this.code = code;
        }

        public MyUnCheckedCustomException(Throwable cause, ErrorCode code) {
                super(cause);
                this.code = code;
        }

        public ErrorCode getCode() {
                return this.code;
        }
}
```

CUSTOM EXCEPTION

```
public void method() {
		try {
			myException(new String("99999999"));
		} catch (MyCustomException e) {
			// handle exception
			log.error(e);
		}
}

//no need throws
private void method(String input) {
		try {
			// do something
		} catch (NumberFormatException e) {
			throw new MyUncheckedCustomException("A message that describes the error.", e,
		ErrorCode.INVALID_INPUT);
		}
}
```

PRINTSTACKTRACE

- It is useful to diagnosing exceptions and it is method of Throwable class which prints the details like class name and line number and helps to trace the code (printStackTrace());

```
try{
Throw new NullPointerException();
}catch(NullpointerException e){
  System.out.println(e);
}

try{
Throw new NullPointerException();
}catch(NullpointerException e){
  e.printStackTrace()
}
```

BEST PRACTICES AND MISTAKES

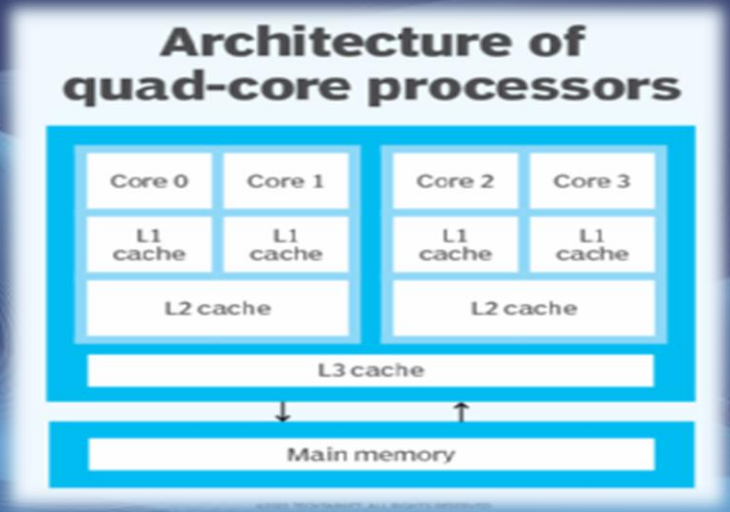
https://stackify.com/best-practices-exceptions-java/
https://stackify.com/common-mistakes-handling-java-exception/

Thread :

- It is a basic unit of CPU Utilization
- It comprises of ThreadId, A program Counter, A register set, and A stack
- It shares with others belonging to the same process its code section, data section, files and signals so on
- A traditional threads or heavy weight process will be having single thread
- A multi thread can perform multiple tasks

Why Multithreading? :

- Rresponsiveness – keep some threads blocked but CPU will be shared for active thread.
- Resource sharing – several threads shares memory with the same address space
- Economical – By allowing context switch across same address spaces and it will reduce the cost rather individual
- Utilization of multiprocessor architectures – Threads will be running in parallel on different processors. It increases concurrency. A single thread process runs only on one CPU.



Multithreading:

- Multithreading in java is a process of executing multiple threads simultaneously.
- Both multiprocessing and multithreading are used to achieve multitasking.

Process	Threads
A process is a collection of one or more threads and associated system resources.	Threads are light-weight processes within a process .
Process can be divided into multiple threads	Threads cannot be sub divided.
Each process has its own memory space	Threads of the same process share a common memory space
It is difficult to create a process	It is easy to create a thread.

Multitasking is a process of executing multiple tasks simultaneously.

The multitasking types are:

1) Process-based Multitasking (Multiprocessing):
   - Each process have its own address in memory i.e. each process allocates separate memory area.
   - Process is heavyweight.
   - Cost of communication between the process is high.
   - Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

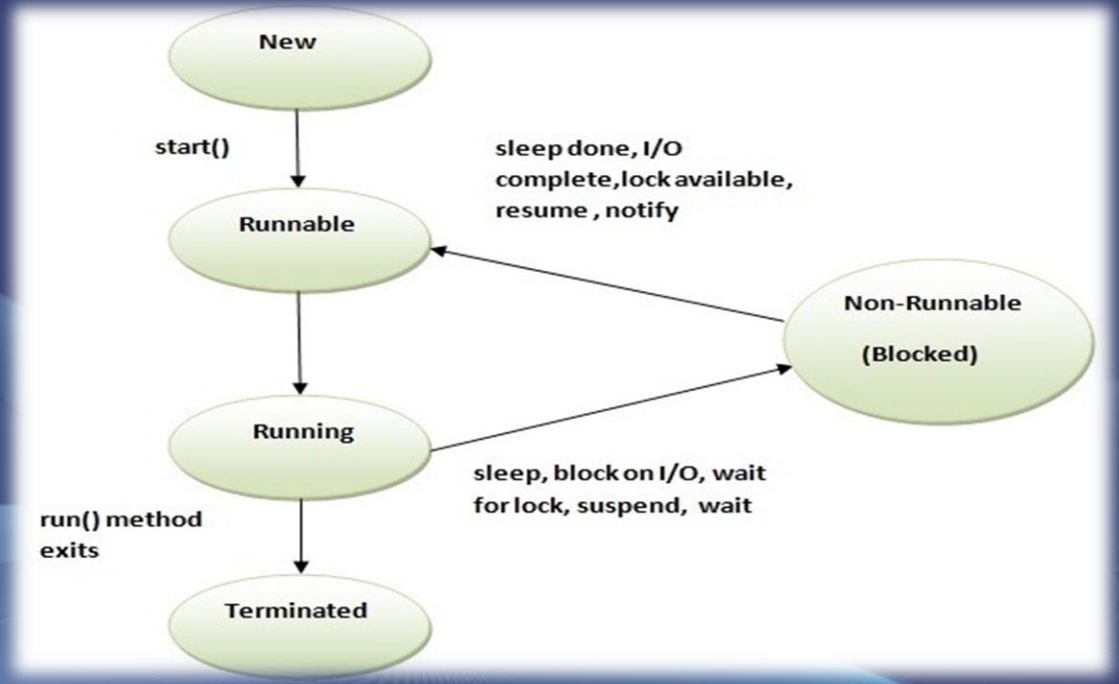
2) Thread-based Multitasking (Multithreading):
   - Threads share the same address space.
   - Thread is lightweight.
   - Cost of communication between the thread is low.

Life cycle of the thread:

During the life time of a thread, there are many states it can enter.

They include :

- New
- Runnable
- Running
- Non-Runnable
- Terminated



Creation of Thread:

By extend thread class	By implementing Runnable interface

class Test **extends** Thread
{
public void run() {
Sytem.out.println(" running..."); }
public static void main(String args[])
{
Test t1=**new** Test();
t1.start();
}
}
 	
// O/P - running…
	class Test2 **implements** Runnable
	{
	public void run() {
	System.out.println("running...");
	}
	public static void main(String args[]) {
	Test2 m1=**new** Test2();
	Thread t1 =**new** Thread(m1);
	t1.start();
	}
	}
	O/P - running...

Method used in Thread:

sleep()	join()	Yield()
Used to sleep a thread for the specified amount of time.	join() waits for a thread to die.	Yield() causes the current thread to temporarily pause its execution
Syntax: 1.public static void sleep(long miliseconds) throws InterruptedException 2. public static void sleep(long miliseconds, int nanos) throws InterruptedException	Syntax: 1.public static join()throws InterruptedException 2.public static join()(long miliseconds) throws InterruptedException	Syntax: 1. public static native void yield()

Naming Thread :

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method.

The syntax of setName() and getName() methods are:

1. public String getName(): used to return the name of a thread.
2. public void setName(String name): used to change the name of a thread.

Thread Priority:

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

The Thread class defines several priority constants:

1. MIN_PRIORITY=1
2. NORM_PRIORITY=5
3. MAX_PRIORITY=10

The default setting is NORM_PRIORITY

Synchronization:

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Synchronization is better in case we want only one thread can access the shared resource at a time.
- Synchronization is mainly used to prevent thread interference and consistency problem.

There are two types of synchronization:

1. Process Synchronization
2. Thread Synchronization

Thread synchronization:

Types of Thread synchronization:

1. Mutual Exclusive

   It helps to keep threads from interfering with one another while sharing data.

   a) Synchronized method
   b) Synchronized block
   c) Static synchronization

2. Co-Operation(Inter-thread communication)

Example for Synchronized Method:

```
class Table{
 synchronized void printTable(int n) {
   for(int i=1;i<=5;i++)
            {       System.out.println(n*i);
     try {
Thread.sleep(400);
    } catch(Exception e) {
System.out.println(e);
}  }  } }
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){  this.t=t;  }
             public void run() {
t.printTable(5);      }  }
public class Demo{
public static void main(String args[]){
Table obj = new Table(); //only one object
MyThread1 t1=new MyThread1(obj);
t1.start();
}  }
```

Synchronized method	Synchronized block
If declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.	Synchronized block can be used to perform synchronization on any specific resource of the method.

Syntax:
Synchronized void methodname()
	Syntax:
	synchronized (object reference expression)
	{
	//code block
	}

Static synchronization:

To make any static method as synchronized, the lock will be on the class not on object.

Deadlock:

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Inter-thread Communication / Co-operation:

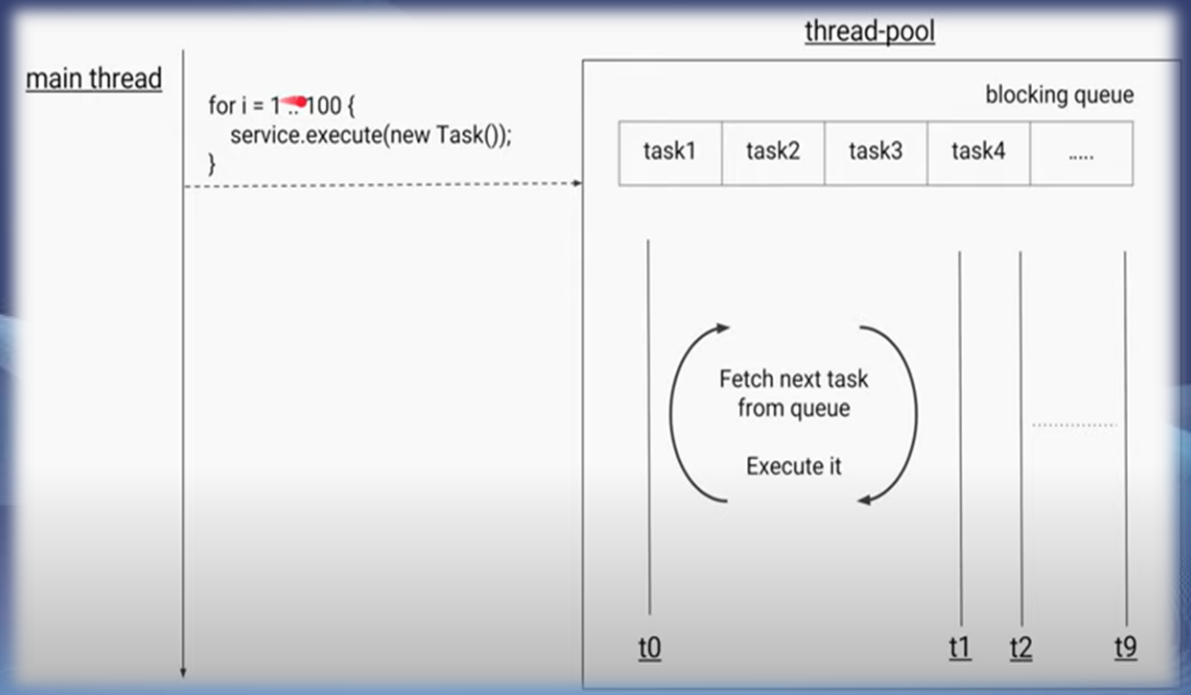
- It is all about allowing synchronized threads to communicate with each other.
- It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of Object class:
  - wait()
  - notify()
  - notifyAll()

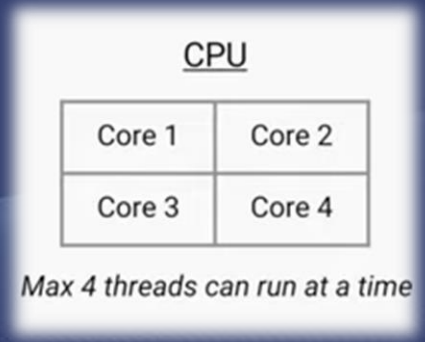
Wait()	Notify()	NotifyAll()
Suspends execution of current thread	It moves only one waiting thread from waiting state to runnable state	It moves all waiting thread from waiting state to runnable state
Only used in synchronised context	Only used in synchronised context	Only used in synchronised context
It releases lock	It will give up the lock	It will give up the lock

ThreadPool:

- Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.
- The Thread Pool pattern helps to save resources in a multithreaded application and to contain the parallelism in certain predefined limits.

.

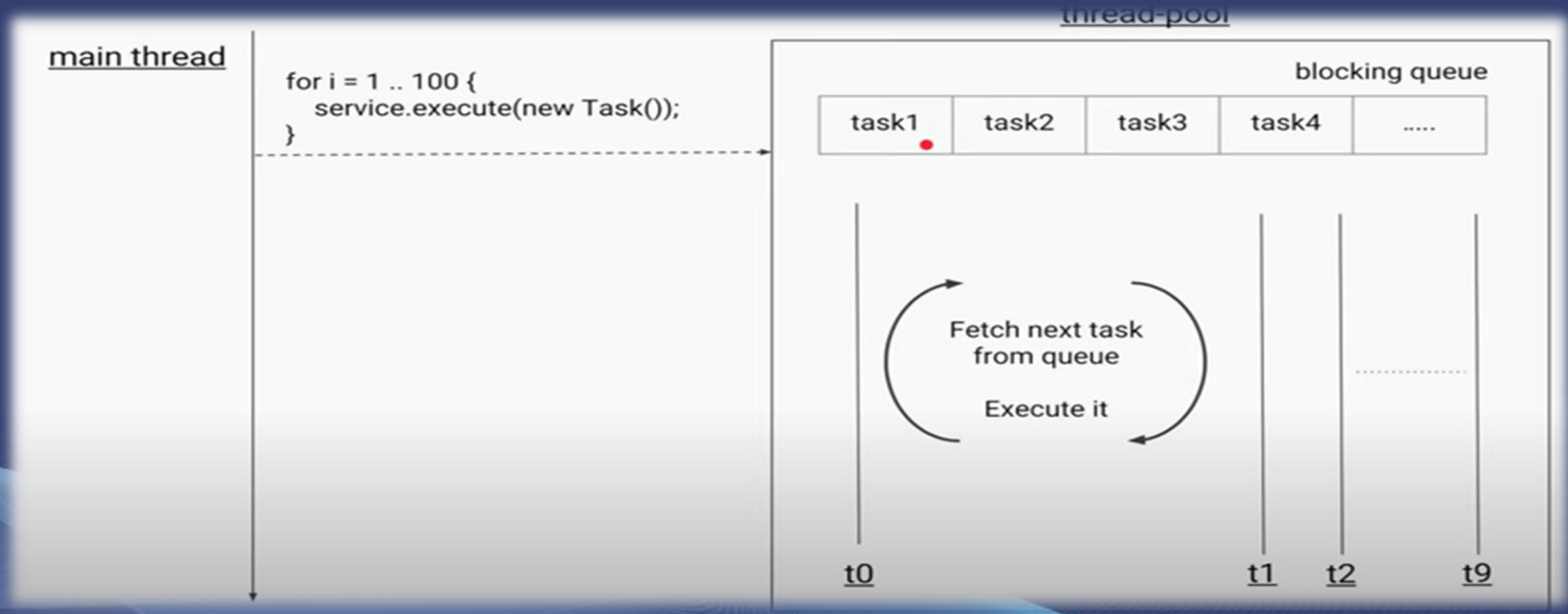




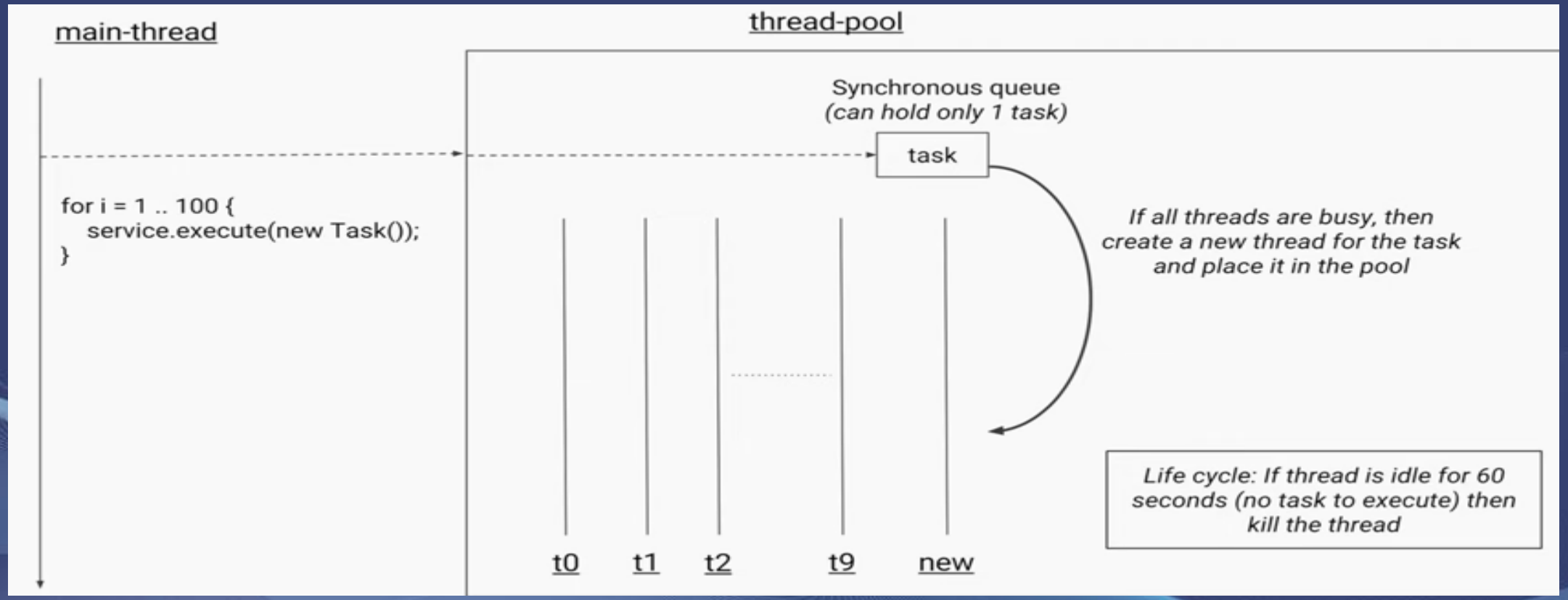
ThreadPool Types:

- FixedThreadPool
- CachedThreadPool
- ScheduledThreadPool
- SingleThreadedExecutorn

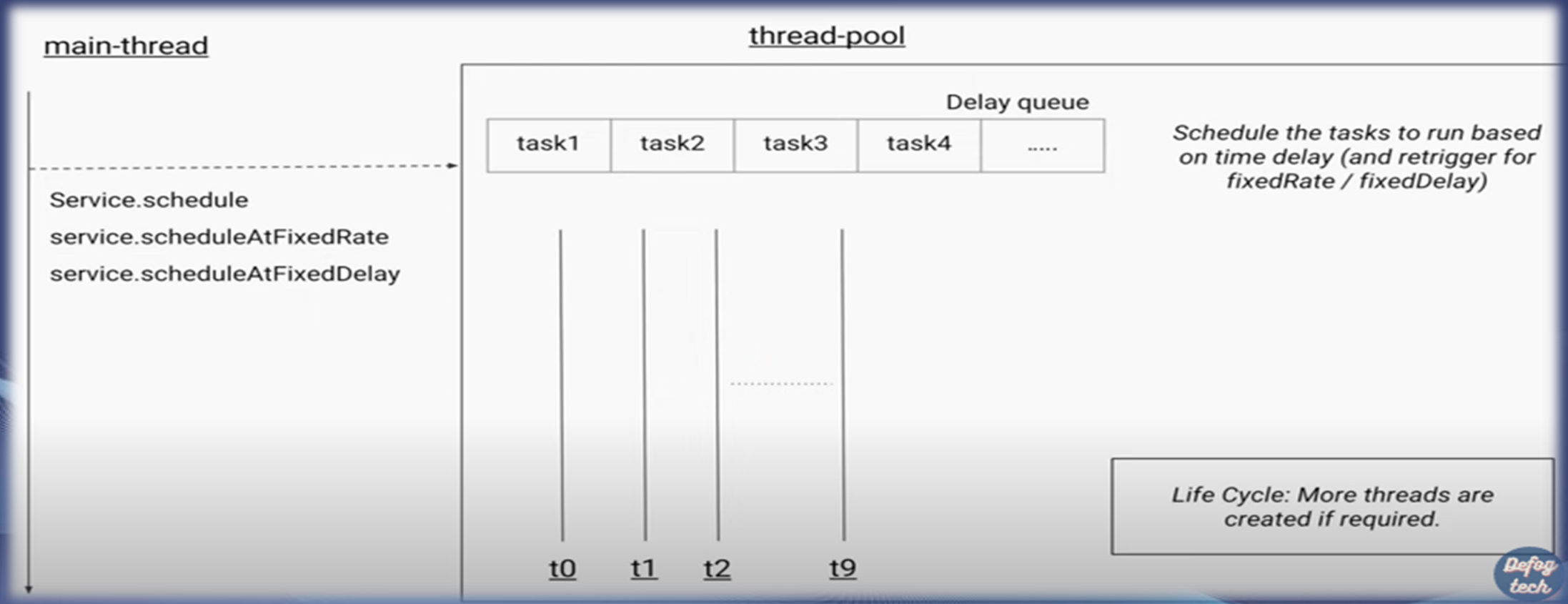
ThreadPool : FixedThreadPool



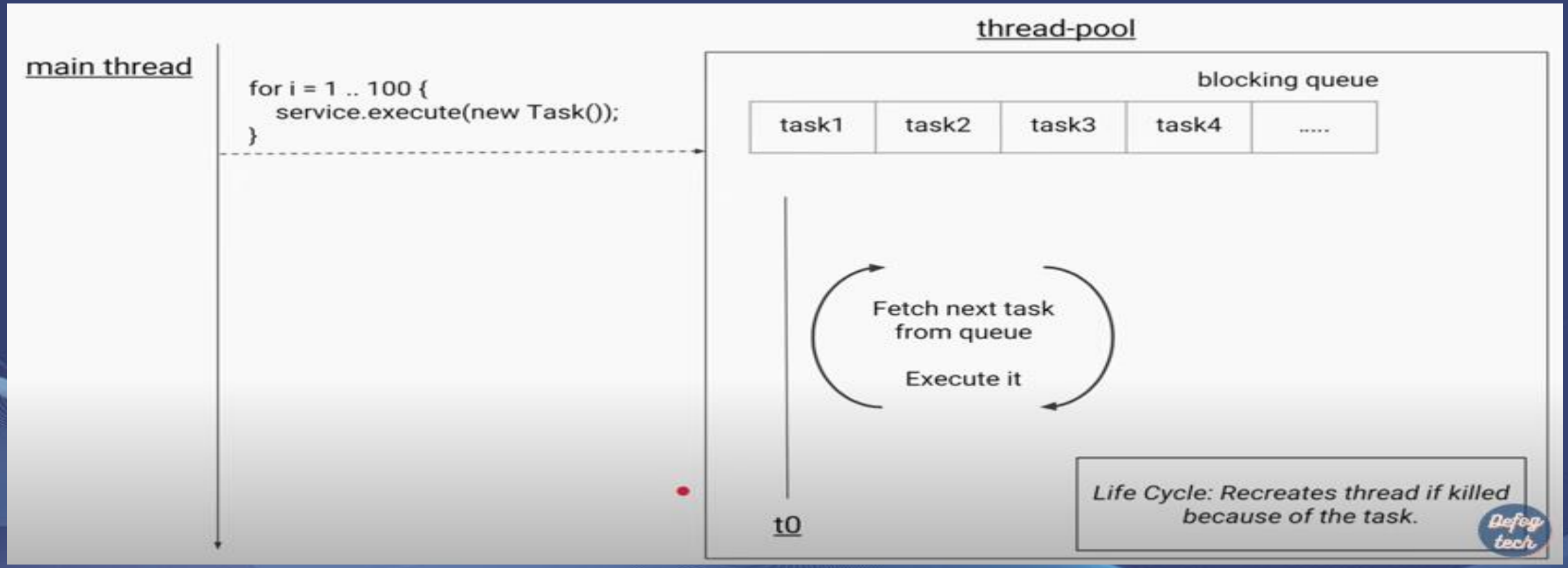
ThreadPool : CachedThreadPool



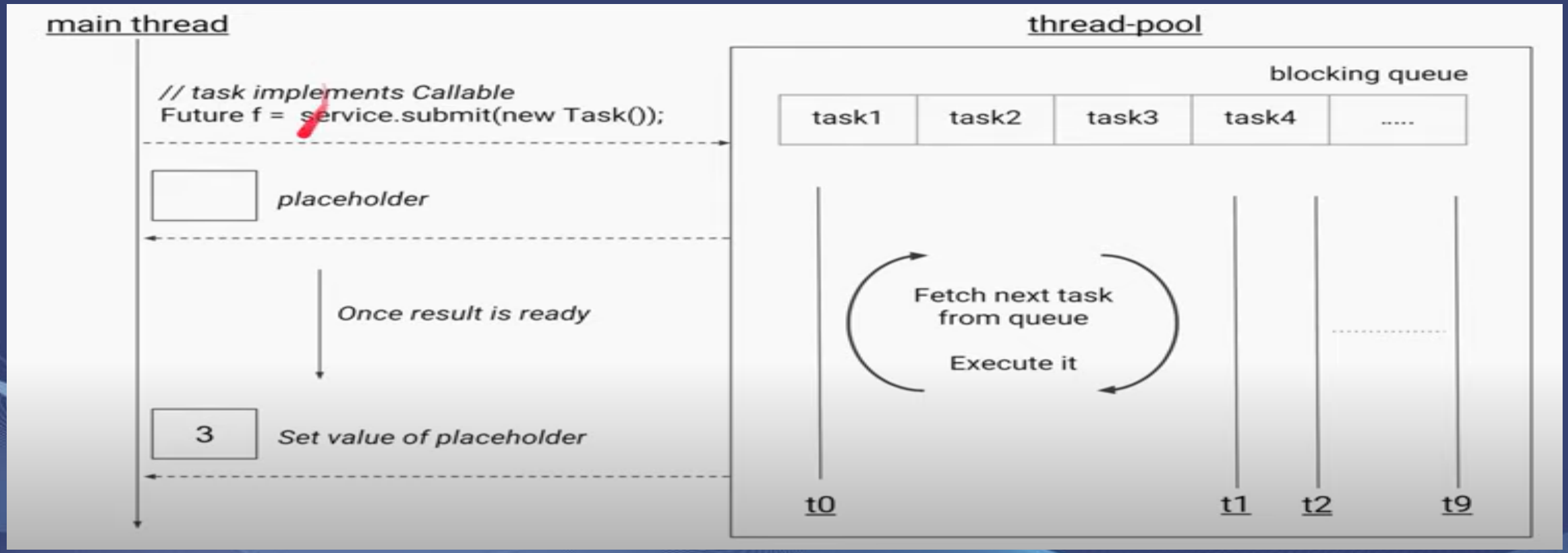
ThreadPool : ScheduledThreadPool



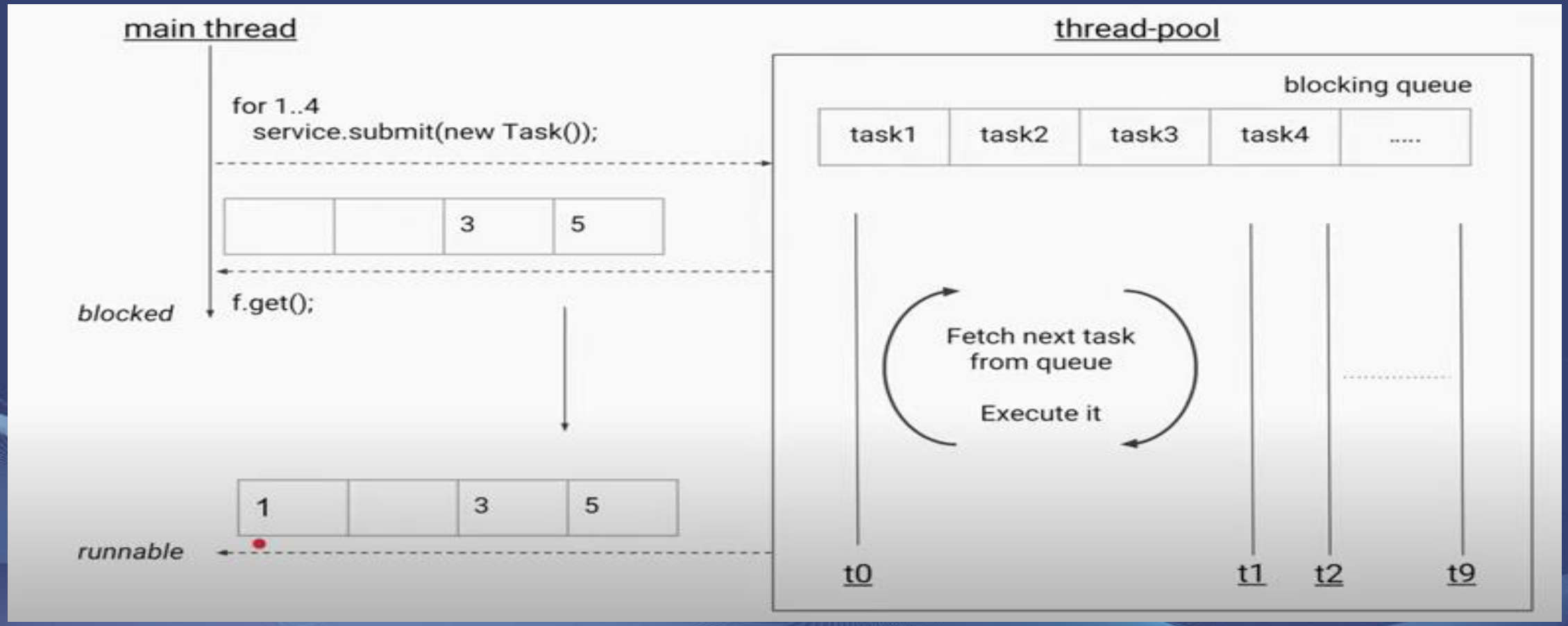
ThreadPool : SingleThreadExecutor



Callable/Future

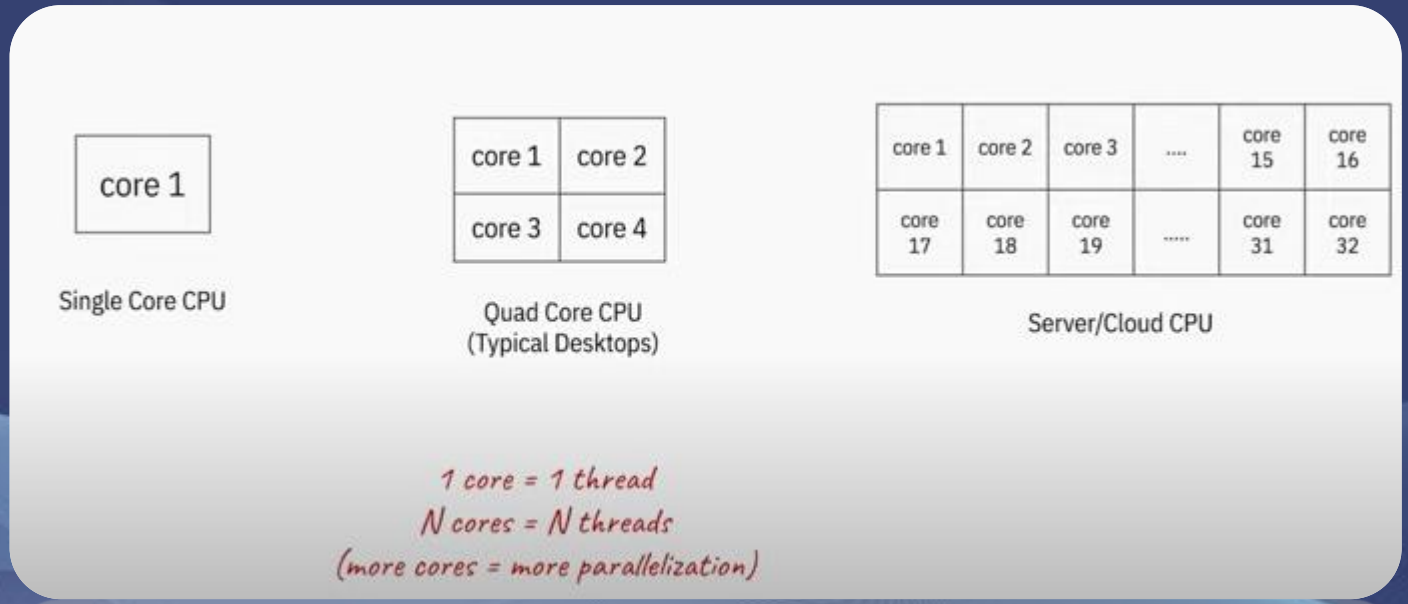


Callable/Future



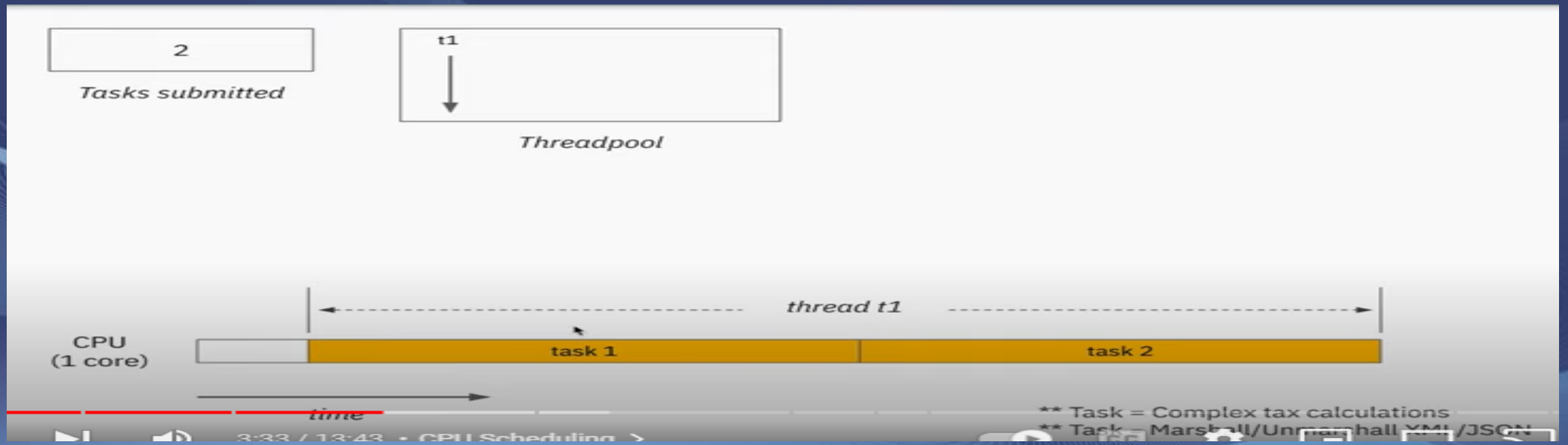
OPTIMAL THREAD POOL SIZE:

- There are different types of Thread pool and ideally this is not just numeric no
- The factors on which Thread pool defined are
  - How many no. of CPU Cores your application has access

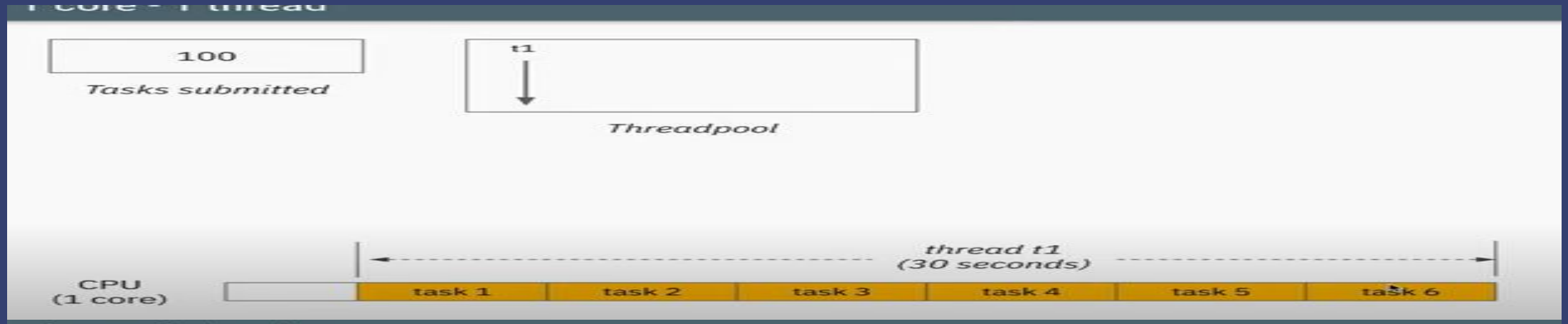


  - Types of Tasks on thread pool
    - CPU Intensive/IO bound

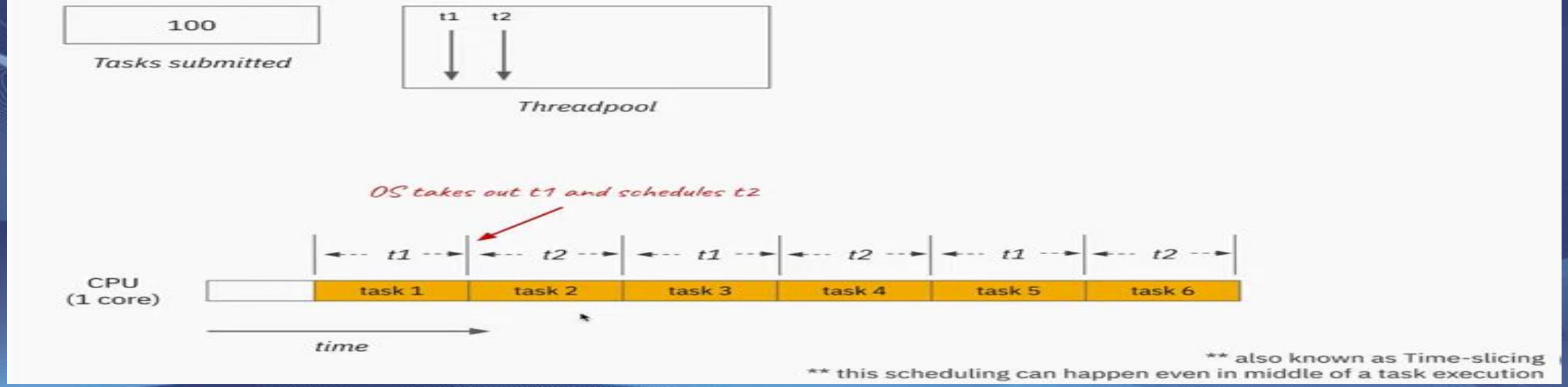
OPTIMAL THREAD POOL SIZE:



OPTIMAL THREAD POOL SIZE:

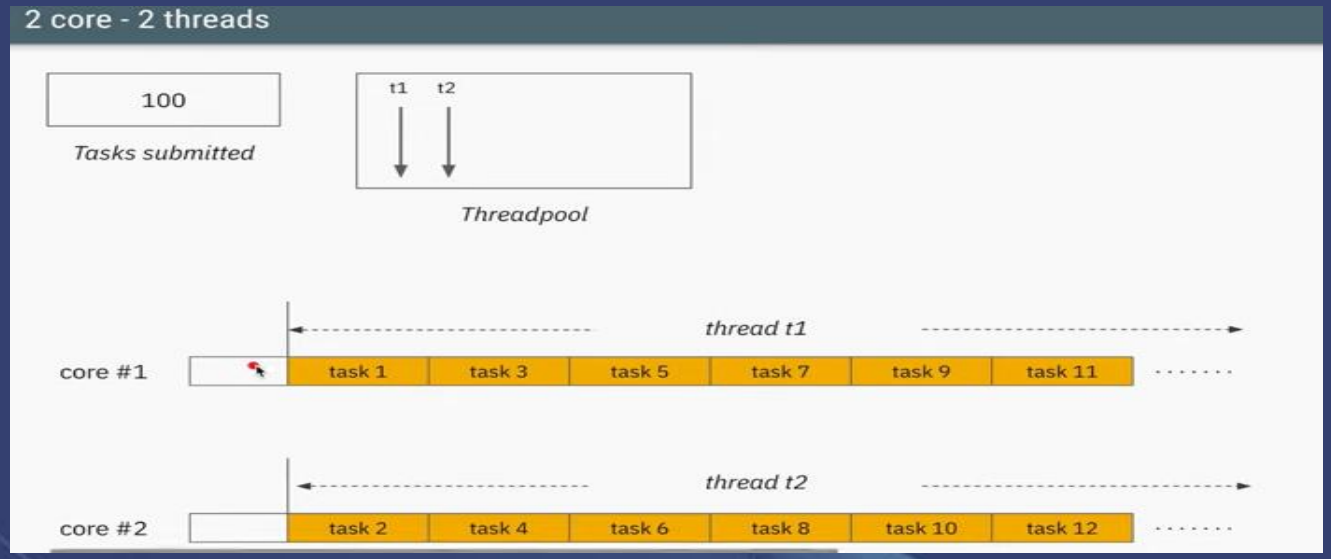


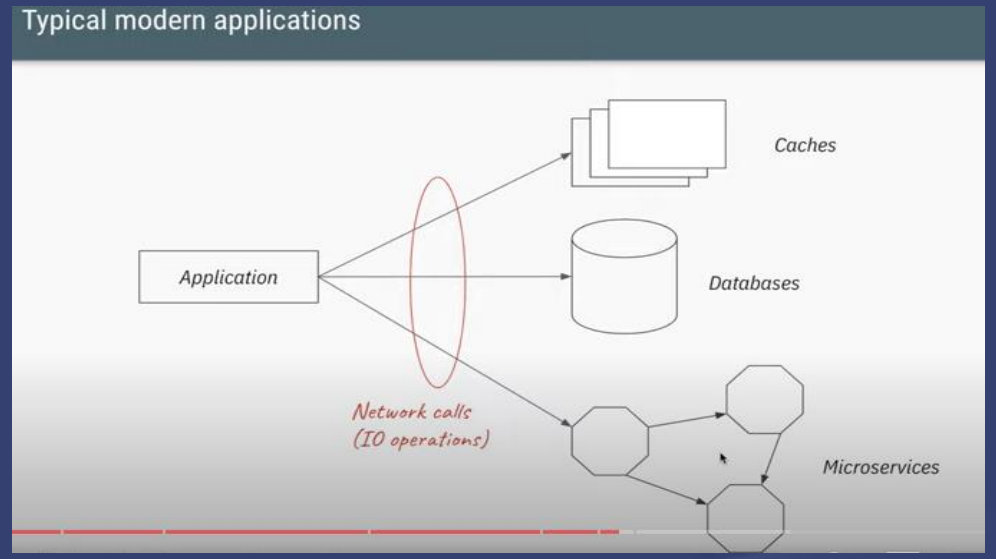
1 core - 2 threads

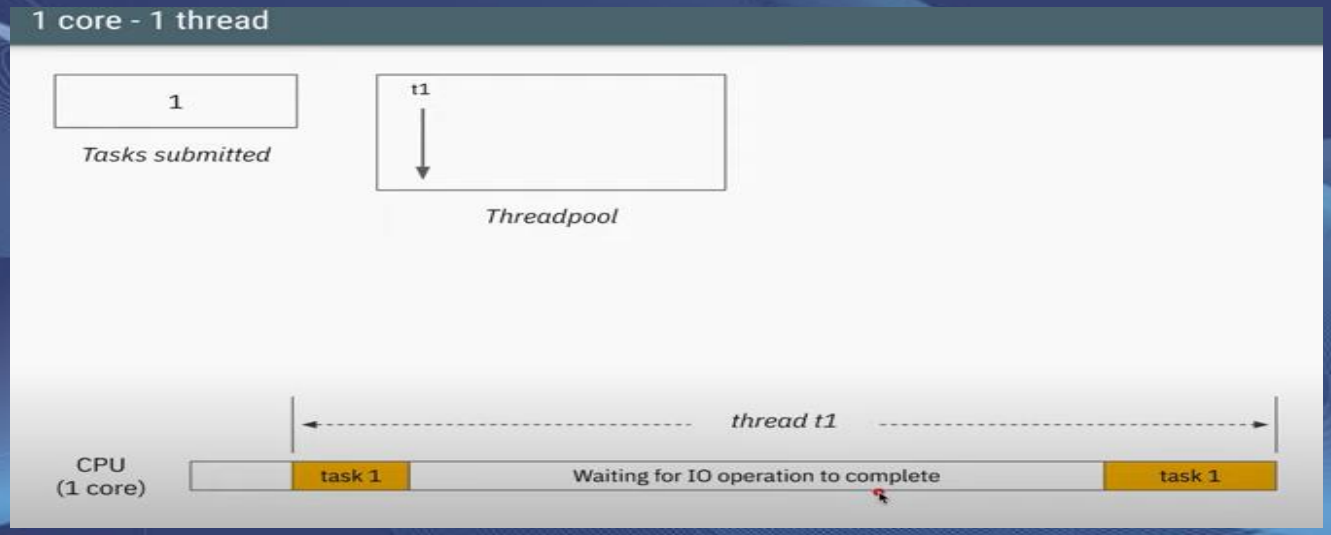


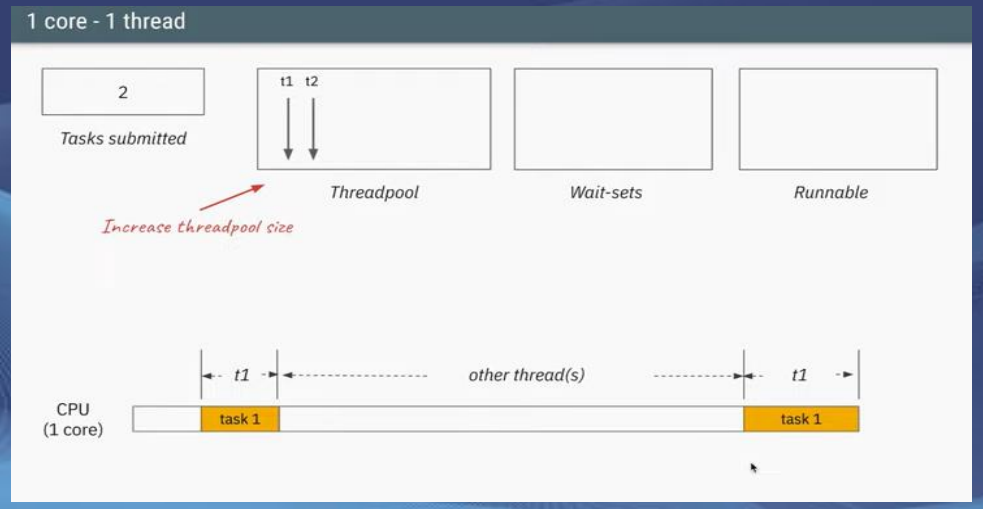
** also known as Time-slicing
** this scheduling can happen even in middle of a task execution

OPTIMAL THREAD POOL SIZE:









OPTIMAL THREAD POOL SIZE:

IDEAL THREAD POOL COUNT = No. of core * [1 + WAITING TIME/CPU TIME]

Other factors:

- Are there other applications running in CPU
- Are there other executive services or threads running on same JVM/Applications
- Too many threads also takes time because of context switch time

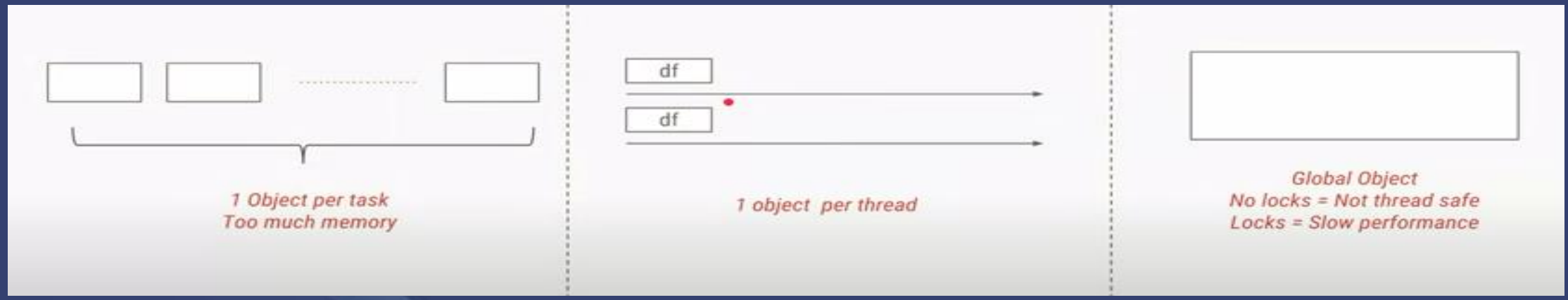
THREAD LOCAL:

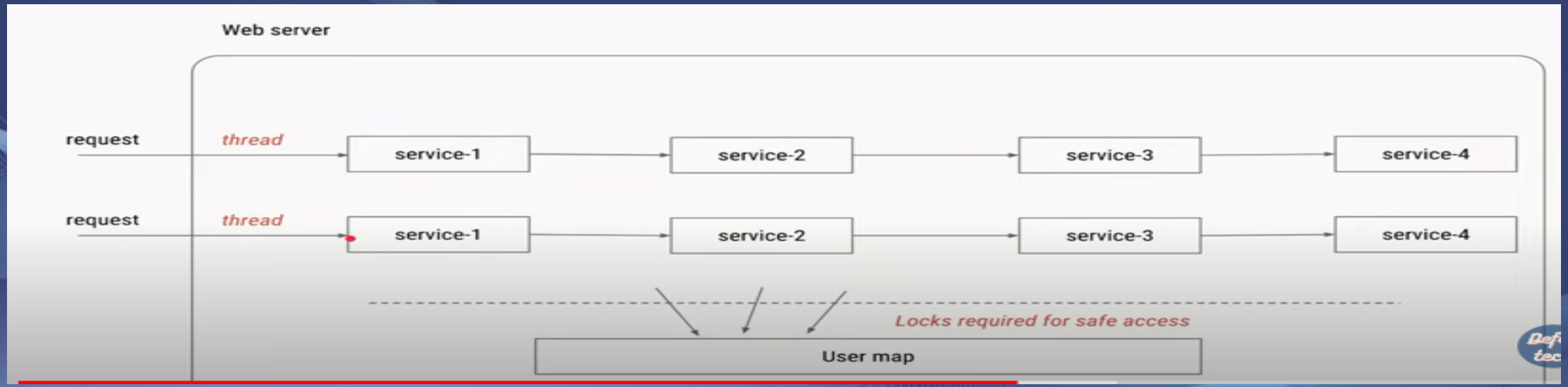
- Thread safe across the threads having a local copy to each thread
- Variables of each thread cannot see each other even though both shares the same code

```
private ThreadLocal threadLocal = new ThreadLocal();

threadLocal.set("thread local value") // assigning value

String threadLocalvalue = (String) threadLocal.get();
```

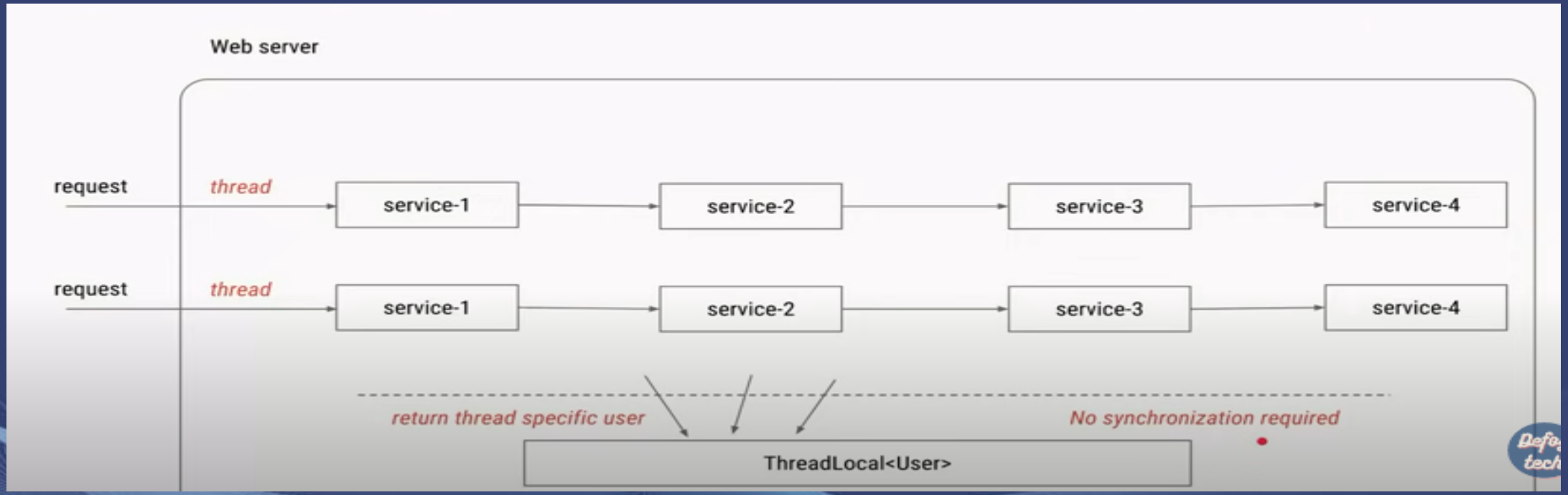
- get() : Returns the value in the current thread's copy of this thread-local variable.
- initialValue() : Returns the current thread's "initial value" for this thread-local variable.
- remove() : Removes the current thread's value for this thread-local variable.
- set(T value) : Sets the current thread's copy of this thread-local variable to the specified value.

THREAD LOCAL:





THREAD LOCAL:



LOCK AND UNLOCK:

a lock is a more flexible and sophisticated thread synchronization mechanism than the standard synchronized block.

Lock interface has been around since Java 1.5. It's defined inside the java.util.concurrent.lock package, and it provides extensive operations for locking

Synchronization	Lock
it completely applied to method or continuous statements	Lock and Unlock API Separately
Synchronized block doesn't support fairness and no support of preference	we can achieve through fairness property and we can ensure longest waiting thread can get priority
A Thread in waiting state of synchronized block can't be interrupted	But, LockAPI Provides lockInterruptibly() interrupt a thread when it is in waiting state.
a thread get blocked sometime can't get back access to synchroized block	lock api provided trylock() method can acquires a lock and reduces waiting time

LOCK AND UNLOCK:

- void lock() – Acquire the lock if it's available. If the lock isn't available, a thread gets blocked until the lock is released.
- void lockInterruptibly() – This is similar to the lock(), but it allows the blocked thread to be interrupted and resume the execution through a thrown java.lang.InterruptedException.
- boolean tryLock() – This is a nonblocking version of lock() method. It attempts to acquire the lock immediately, return true if locking succeeds.
- boolean tryLock(long timeout, TimeUnit timeUnit) – This is similar to tryLock(), except it waits up the given timeout before giving up trying to acquire the Lock.
- void unlock() unlocks the Lock instance.

LOCK AND UNLOCK:

```
Lock lock = ...;
lock.lock();
 try {   // access to the shared resource
         }
finally {
          lock.unlock(); }
```

- In addition to the Lock interface, we have a ReadWriteLock interface that maintains a pair of locks, one for read-only operations and one for the write operation. The read lock may be simultaneously held by multiple threads as long as there is no write.
- ReadWriteLock declares methods to acquire read or write locks:
- Lock readLock() returns the lock that's used for reading.
- Lock writeLock() returns the lock that's used for writing.

ReentrantLock:

ReentrantLock class implements the Lock interface. It offers the same concurrency and memory semantics as the implicit monitor lock accessed using synchronized methods and statements, with extended capabilities.

```
public class SharedObject {
//... ReentrantLock lock = new ReentrantLock();
int counter = 0;
public void perform() {
lock.lock();
try { // Critical section here count++; }
 finally { lock.unlock(); } } //... }
```

ReentrantLock:

- we are wrapping the lock() and the unlock() calls in the try-finally block to avoid the deadlock situations.
- the thread calling tryLock() will wait for one second and will give up waiting if the lock isn't available.

```
public void performTryLock(){
 //... boolean isLockAcquired = lock.tryLock(1, TimeUnit.SECONDS);
 if(isLockAcquired) {
 try { //Critical section here }
finally { lock.unlock(); } } //...
 }
```

ReentrantReadWriteLock:

ReentrantReadWriteLock class implements the ReadWriteLock interface

- Read Lock – If no thread acquired the write lock or requested for it, multiple threads can acquire the read lock.
- Write Lock – If no threads are reading or writing, only one thread can acquire the write lock.

```
Lock readLock = lock.readLock();

//...

 public String get(String key){

try { readLock.lock();

return syncHashMap.get(key); } finally { readLock.unlock();

} }
```

ReentrantReadWriteLock:

```
public class SynchronizedHashMapWithReadWriteLock {

Map<String,String> syncHashMap = new HashMap<>();

 ReadWriteLock lock = new ReentrantReadWriteLock();

 Lock writeLock = lock.writeLock();

public void put(String key, String value) {

 try { writeLock.lock();  syncHashMap.put(key, value); }

 finally { writeLock.unlock(); }  }

public String remove(String key){ try { writeLock.lock();

return syncHashMap.remove(key);

 } finally { writeLock.unlock(); } } //... }
```

DIFFERNCE BETWEEN LOCK AND SYNCHRONIZATION:

	Synchroinization	Lock
Acquire Lock	just synchronized key word sufficient	Reentrant class provides lock() method
Release Lock	Implicitly	unLock() method should be called
interrupt	Not possible	lockInterrupibly to interupt
Fairness	it doesn't gurantee for longest waiting thread	Possible to allocate longest waiting thread
order of release	it releases as in the same order of qcquired	It can be any order

https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/

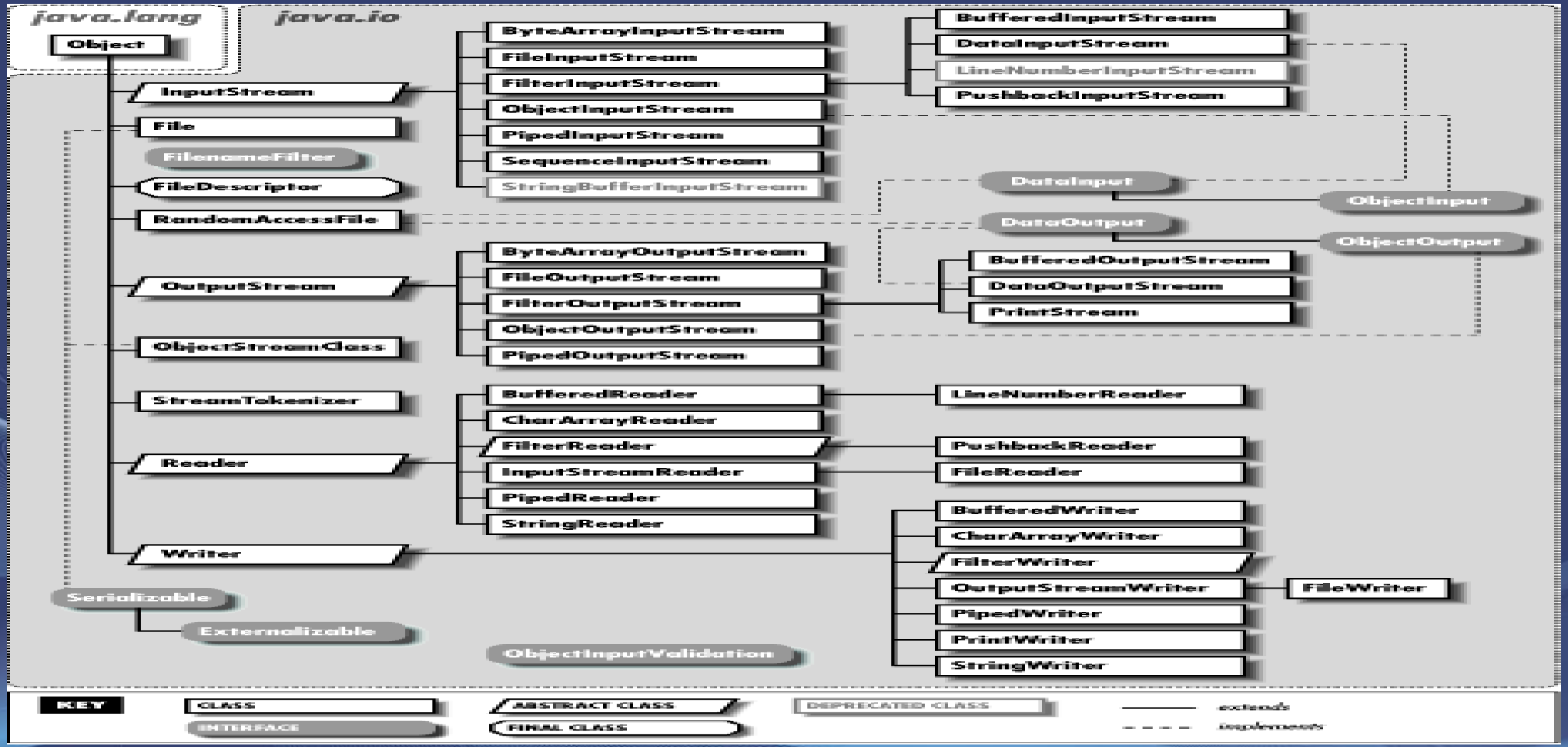
IOSTREAMS:

- An I/O Stream represents an input source or an output destination
- A stream can represent many different kinds of sources and destinations
  - disk files, devices, other programs, a network socket, and memory arrays
- Streams support many different kinds of data
  - simple bytes, primitive data types, localized characters, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways
- No matter how they work internally, all streams present the same simple model to programs that use them

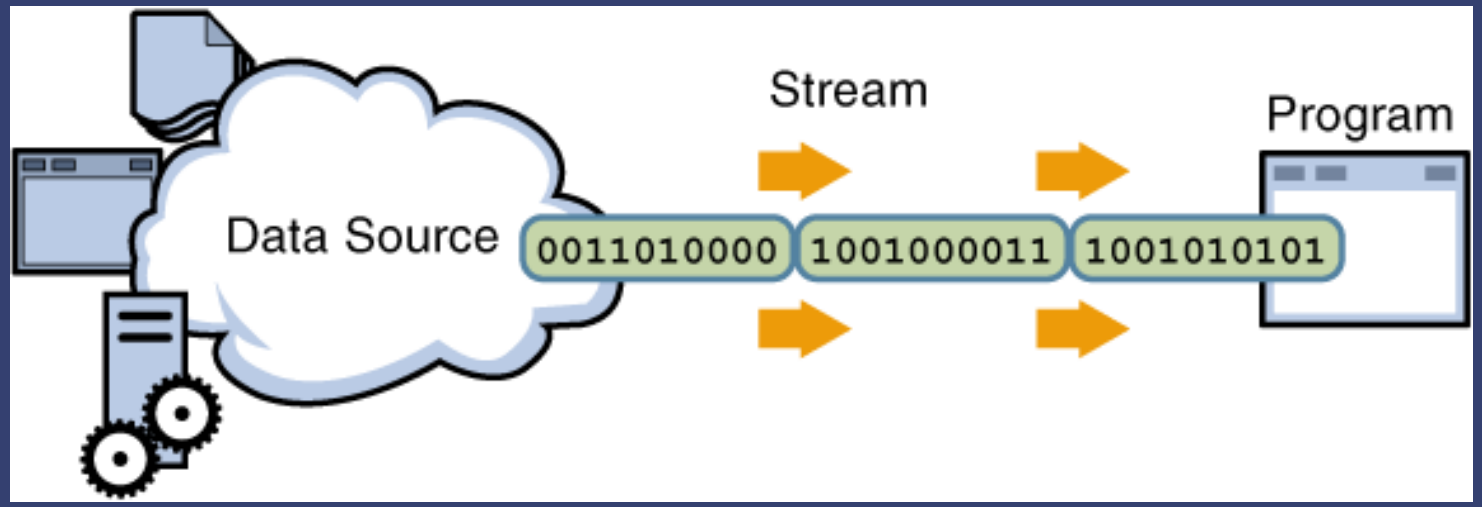
  – A stream is a sequence of data

https://docs.oracle.com/javase/7/docs/api/help-doc.html

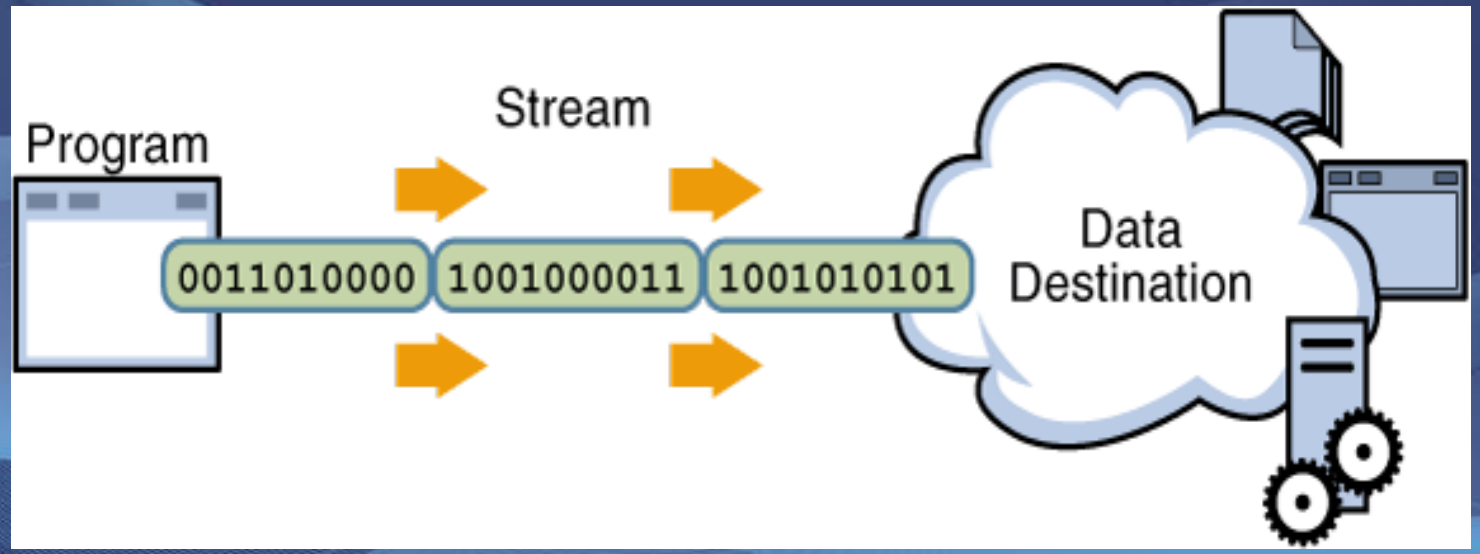
IOSTREAMS


java.lang
java.io
Object
InputStream
File
FilenameFilter
FileDescriptor
RandomAccessFile
OutputStream
ObjectStreamClass
StreamTokenizer
Reader
Writer
Serializable
Externalizable
ObjectInputValidation
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
DataInput
ObjectInput
DataOutput
ObjectOutput
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
PipedReader
StringReader
LineNumberReader
PushbackReader
FileReader
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
PipedWriter
PrintWriter
StringWriter
FileWriter
KEY
CLASS
INTERFACE
ABSTRACT CLASS
FINAL CLASS
DEPRECATED CLASS
extends
implements

INPUT STREAM:



OUTPUT STREAM



GENERAL IOSTREAMS TYPES

- *Character and Byte Streams*
  - *Character vs. Byte*
- *Input and Output Streams*
- *– Based on source or destination*
- *Node and Filter Streams*
- *– Whether the data on a stream is manipulated or transformed or not*
- *CONTROL FLOW*
  - *Create a stream object and associate it with a datasource*
  - *(data-destination)*
  - *Give the stream object the desired functionality*
  - *through stream chaining*
  - *while (there is more information)*
  - *read(write) next data from(to) the stream*
  - *close the stream*

GENERAL IOSTREAMS TYPES

- *Character and Byte Streams*
  - *Character vs. Byte*
- *Input and Output Streams*
- *– Based on source or destination*
- *Node and Filter Streams*
- *– Whether the data on a stream is manipulated or transformed or not*
- *CONTROL FLOW*
  - *Create a stream object and associate it with a datasource*
  - *(data-destination)*
  - *Give the stream object the desired functionality*
  - *through stream chaining*
  - *while (there is more information)*
  - *read(write) next data from(to) the stream*
  - *close the stream*

IMP CLASSES

- *FileInputStream, FileOutputStream – Bystream operations*
- *FileReader, FileWriter – character stream classes*
- *DataInputStream, DataOutputStream – created as a wrapper for primitive datas*
- *BufferedInputStream, BufferedOutputStream,BufferedReader, BufferedWriter – Buffering a byte or character streams*
- *File class*

COLLECTION

- A "collection" object — sometimes called a container — is simply an object that groups multiple elements into a single unit
- Collection in java is a framework that provides an architecture to store and manipulate the group of objects.
- Collections are growable in nature.
- Collections can hold both homogeneous and heterogeneous elements.
- Difference b/w Collection and Collections is Collections is an utility class present in java.util.package to define several utility methods (like Sorting, Searching..) for Collection objects.
- Every collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement.

Collection:

- A collections framework is a unified architecture for representing and manipulating collections
- All collections frameworks contain the following:
  - Interfaces
  - Implementations
  - Algorithms

  - https://docs.google.com/spreadsheets/u/0/d/e/2PACX-1vSbnpXFxZWW1p65OvLOMg9ilxTN4oCZl1fV5srNyN4QoKNdLKKs9cmZORviR0Gbz1-RYAzC8QhQDUmj/pubhtml?gid=0&single=true&pli=1

Benefits of Collection Framework

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs

  The collection interfaces are the vernacular by which APIs pass collections back and forth
- Reduce effort to learn and use new APIs
- Reduces effort to design new APIs
- Fosters software reuse
  - New data structures that conform to the standard collection interfaces are by nature reusable

TWO SCHEMES OF TRAVERSING

- for-each

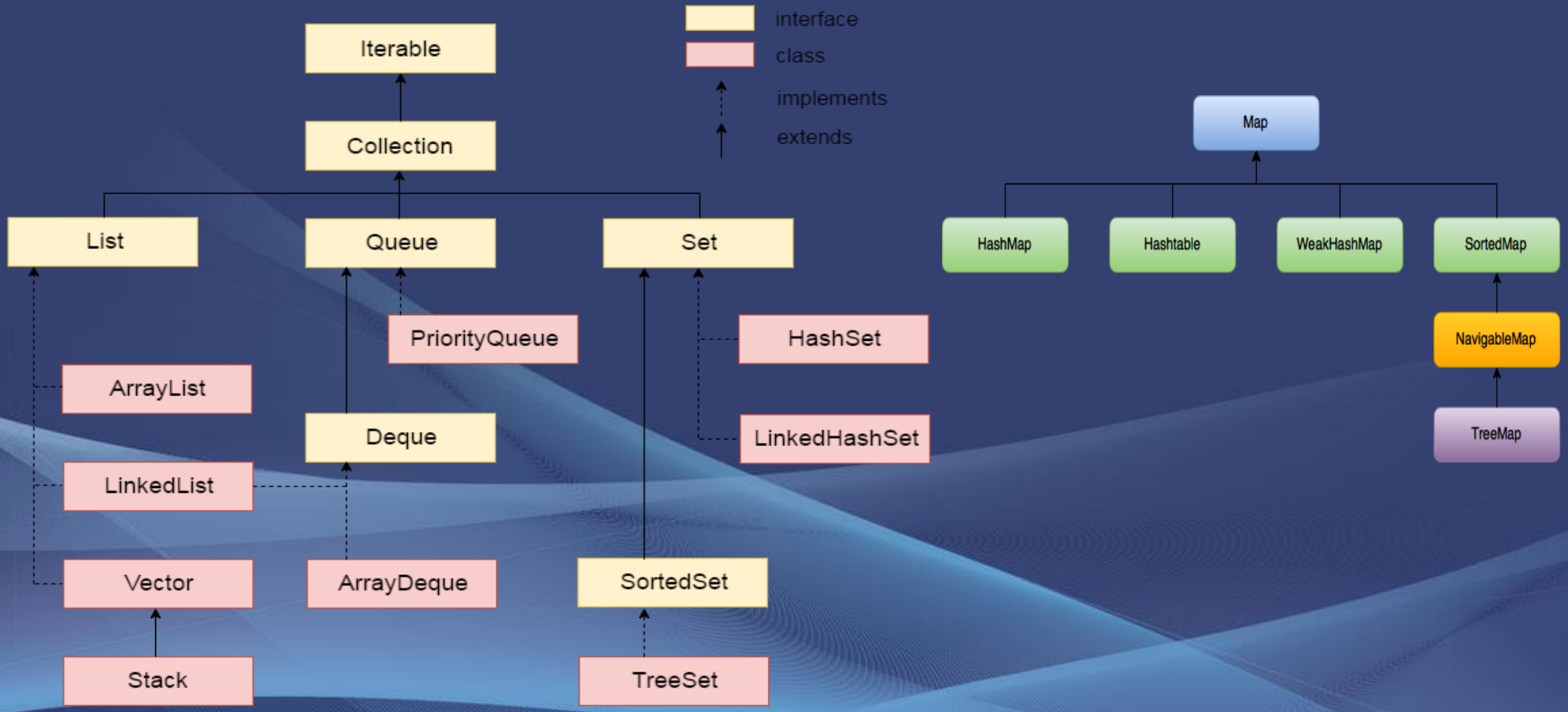
  The for-each construct allows you to concisely traverse a collection or array using a for loop

  ```
  for (Object o: collection)
        System.out.println(o);
  ```

- Iterator

  An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

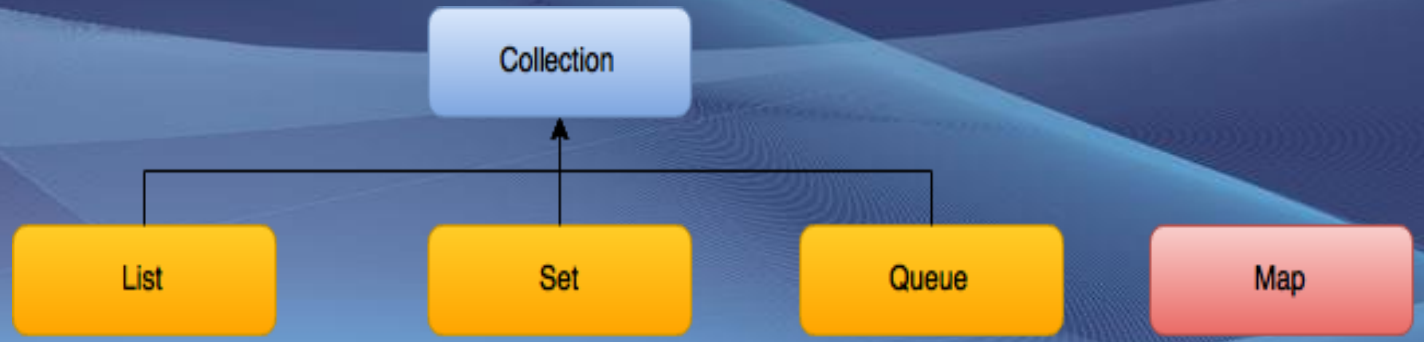
Hierarchy of Collection Framework



Key Interfaces of Collection Framework:

1) Collection Interface :

- Collection represents a single unit of objects. i.e. a group. and To represent a group of individual objects as a single entity.
- Collection interface defines the most common methods which are applicable for any Collection object and is considered as root interface of Collection Framework.



```
interface Collection<E> extends Iterable<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); //optional
boolean remove(Object element); //optional
Iterator<E> iterator();
// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
//optional
boolean removeAll(Collection<?> c); //optional
boolean retainAll(Collection<?> c); //optional
void clear(); //optional
// Array operations
Object[] toArray();
<T> T[] toArray(T[] a);
                                                    }
```

Bulk operations:

containsAll() — returns true if the target Collection contains all of the elements in the specified Collection.
addAll() — adds all of the elements in the specified Collection to the target Collection.
removeAll() — removes from the target Collection all of its elements that are also contained in the specified Collection.
retainAll() — removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
clear() — removes all elements from the Collection.

2) List Interface :

- List interface extends the Collection interface to provides the functionality to store and manage a sequence of items.
- It contains methods to insert and delete elements in index basis. It is a factory of ListIterator interface
- List is useful for represent a group of individual objects as a single entity where duplicates are allowed and insertion order preserved.

- Declaration of List Interface:

  public interface List<E> extends Collection<E>

- Methods of List Interface:

Modifier and Type	Method	Modifier and Type	Method
public void	add(int index, Object element)	public object	remove(int index)
public boolean	addAll(int index, Collection c)	ListIterator	listIterator()
public object	get(int index)	ListIterator	listIterator(int index)
public object	set(int index, Object element)		

ArrayList:

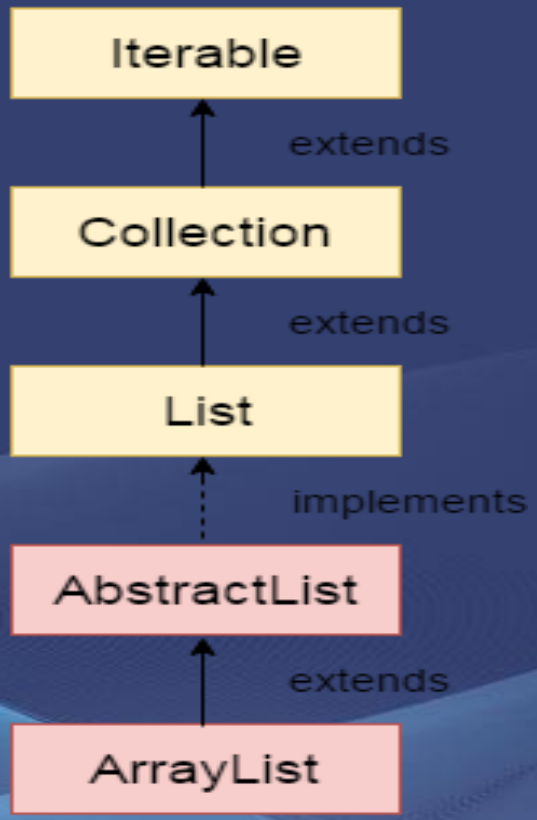
- Java ArrayList class uses a dynamic array for storing the elements.
  - It inherits AbstractList class and implements List interface.
- The important points about Java ArrayList class are:
  - It can contain duplicate elements.
  - It class maintains insertion order.
  - It is non-synchronized.
  - The underlined date structure is Resizable Array or Growable Array.
  - It allows Heterogeneous objects.
  - It allows Random Access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.
- It is efficient for retrieval data not for frequent adding and modification.

Declaration of ArrayList:

public class ArrayList<E> extends AbstractList<E> implements List<E>

Constructors of ArrayList:

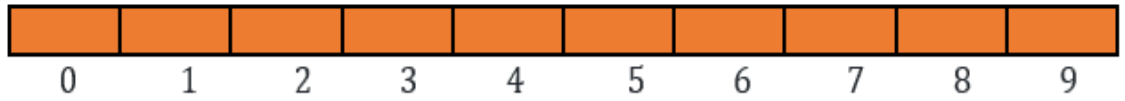
- ArrayList()
- ArrayList(Collection c)
- ArrayList(int capacity)



Hierarchy of ArrayList class

Default size of ArrayList

```
1.  ArrayList<Integer> list = new ArrayList<Integer>();
```




```
1.  ArrayList<Integer> list = new ArrayList<Integer>(15);
```

ArrayList:
Initial Capacity:10
Load Factor:1 (when the list is full)
Growth Rate: current_size + current_size/2
Vector:
Initial Capacity:10
Load Factor:1 (when the list is full)
Growth Rate: current_size * 2 (if capacityIncrement is not defined)
current_size + capacityIncrement (if capacityIncrement is defined during vector initialization)

ArrayList grow factor
newCapacity = oldCapacity + (oldCapacity >> 1

Vector:
newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity

Methods of ArrayList:

Modifier and Type	Method	Modifier and Type	Method
public boolean	addAll(Collection c)	public int	lastIndexOf(Object ele)
public boolean	add(Object ele)	public int	indexOf(Object ele)
public boolean	addAll(int index, Collection c)	public Object[]	toArray()
public void	add(int index, Object element)	public Object[]	toArray(Object[] a)
public void	clear()	public Object	clone()
public void	trimToSize()		

Example for ArrayList:

```
import java.util.*;
Class ArrayListDemo {
public static void main(String args[]){
ArrayList I = new ArrayList();
I.add("A");
I.add(10);
I.add("A");
I.add(null);
System.out.println(I);  // O/P – [A, 10, A, null]
I.remove(2);
System.out.println(I);  // O/P – [A, 10, null]
I.add("2", "B");
I.add("C");
System.out.println(I); // O/P – [A, 10, B, null, C]
}}
```

LinkedList:

- Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

- The important points about Java LinkedList are:
  - It can contain duplicate elements.
  - It maintains insertion order.
  - It is non-synchronized.
  - In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
  - Java LinkedList class can be used as list, stack or queue.

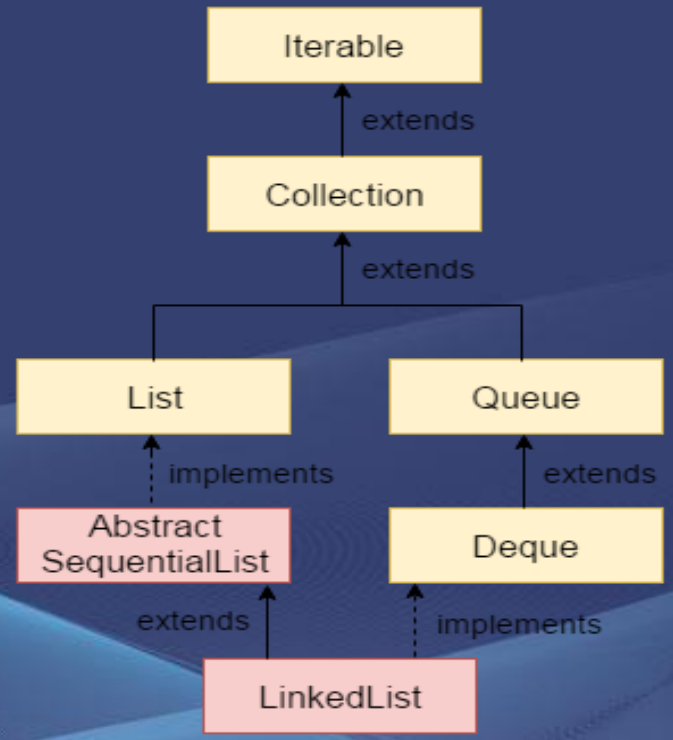
  - Java linkedList is implemented through doubly linked list.

Declaration of LinkedList:

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E> , Deque<E>, Cloneable, Serializable

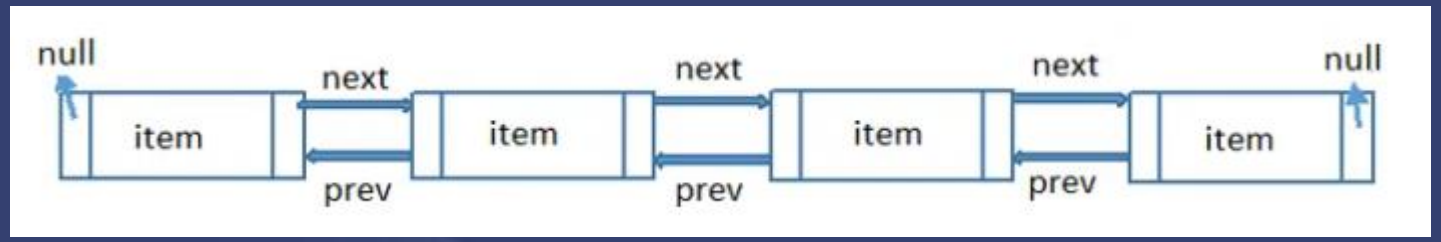
Constructors of LinkedList:

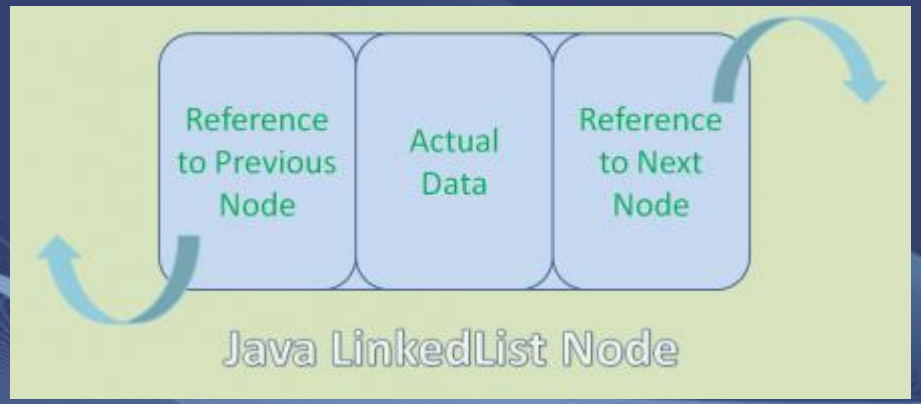
- LinkedList()
- LinkedList(Collection c)

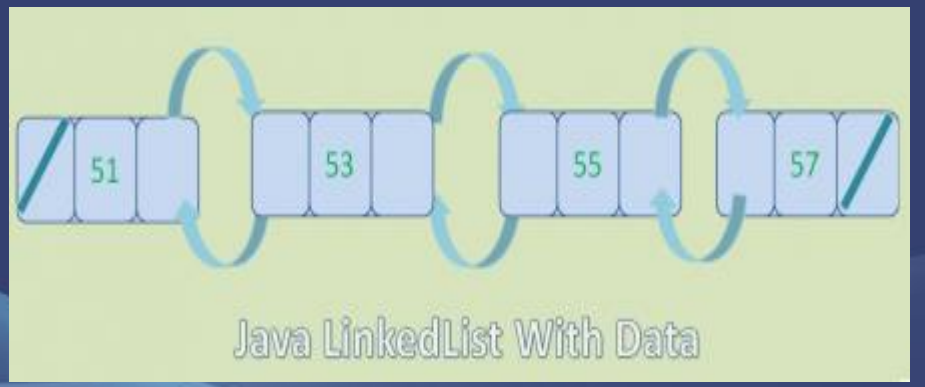


Hierarchy of LinkedList class

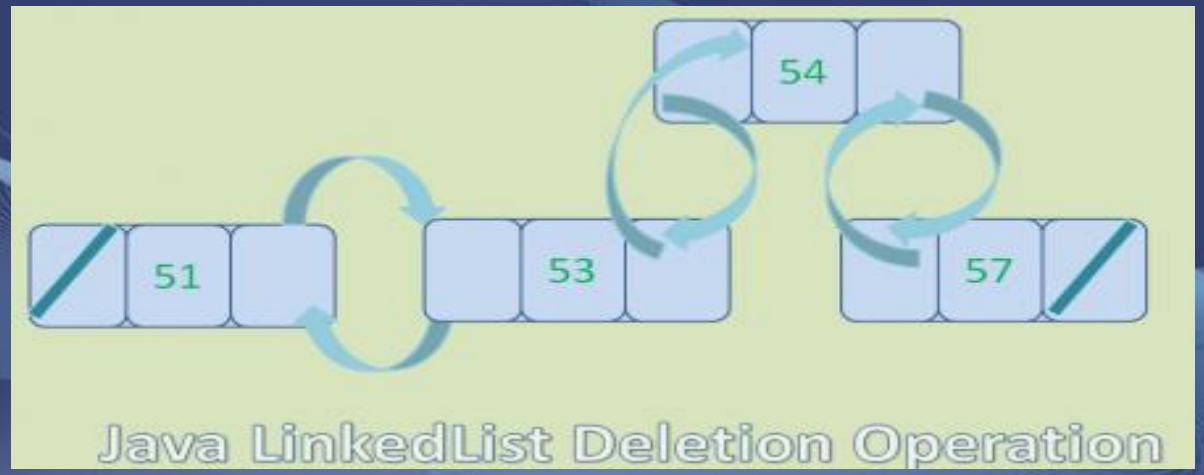
Internals LinkedList:







Internals LinkedList:



Methods of LinkedList:

Modifier and Type	Method	Modifier and Type	Method
public boolean	contains(Object o)	public int	size()
public boolean	add(Object o)	public int	indexOf(Object o)
public boolean	remove(Object o)	public int	lastIndexOf(Object o)
public void	add(int index, Object element)	public Object	getFirst()
public void	addFirst(Object o)	public Object	getLast()
public void	addLast(Object o)		

Example for LinkedList:

```
import java.util.*;
Class LinkedListDemo {
public static void main(String args[]){
List I = new LinkedList();
I.add("A");
I.add(10);
I.add("A");
I.add(null);
System.out.println(I);  // O/P – [A, 10, A, null]
I.remove(2);
System.out.println(I);  // O/P – [A, 10, null]
I.add("2", "B");
I.add("C");
System.out.println(I); // O/P – [A, 10, B, null, C]
}}
```

ArrayList	LinkedList
growable array	Doubly linked list
Random access interface	Do not implement random access interface
best choice whenever we need to access frequently	Best choice whenever the insertion and deletion
worst choice for insertion deletion	worst choice for access items

Vector:

- Vector implements List interface and maintains insertion order.
- Vector is synchronized. Hence Vector class is mainly useful for multithreaded environment, since all the methods implemented in Vector class.
- It allows Duplicate objects, Heterogeneous objects and 'null' insertion.
- The Vector class implements a growable array of objects. Vector doubles the array size if total number of elements exceeds than its capacity.

Declaration of Vector:

```
Vector<String> myVector = new Vector<String>();
```

Constructors of Vector:

- Vector()
- Vector(int size)
- Vector(int size, int incr)
- Vector(Collection c)

Methods of Vector:

Modifier and Type	Method	Modifier and Type	Method
public boolean	add(Object o)	public int	size()
public boolean	remove(Object o)	public int	capacity()
public void	add(int index, Object o)	public Object	get(int index)
public void	addElement(Object o)	public Object	elementAt(int index)
public void	remove(Object o)	public Object	firstElement()
Public void	removeElement(Object o)	public Object	lastElement()

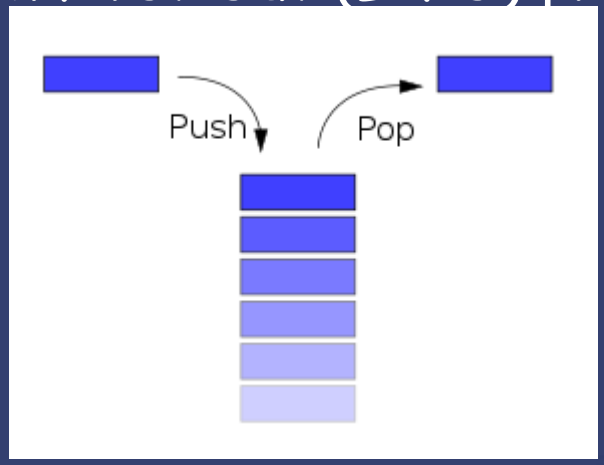
Example for Vector:

```
import java.util.*;
Class VectorDemo {
public static void main(String args[]){
Vector v = new Vector();
System.out.println(v.capacity());  // O/P – [ 10]
for(int i=0;i<10;i++) {
v.addElement(i);
}
System.out.println(v.capacity());  // O/P – [ 10]
v.addElement("A");
System.out.println(v.capacity());  // O/P – [ 20]
v.addElement(v);
}}
```

Vector	ArrayList
synchronized	not sychroinized
increments 100% of the capacity to grow size	increments 50% of the the capacity to grow size
legacy class	not legacy class
slow	fast

Stack:

- Stack class is a collection that is based on the Last In First Out (LIFO) principle.



- Constructor of Stack:
  - Stack()
- Methods of Stack:

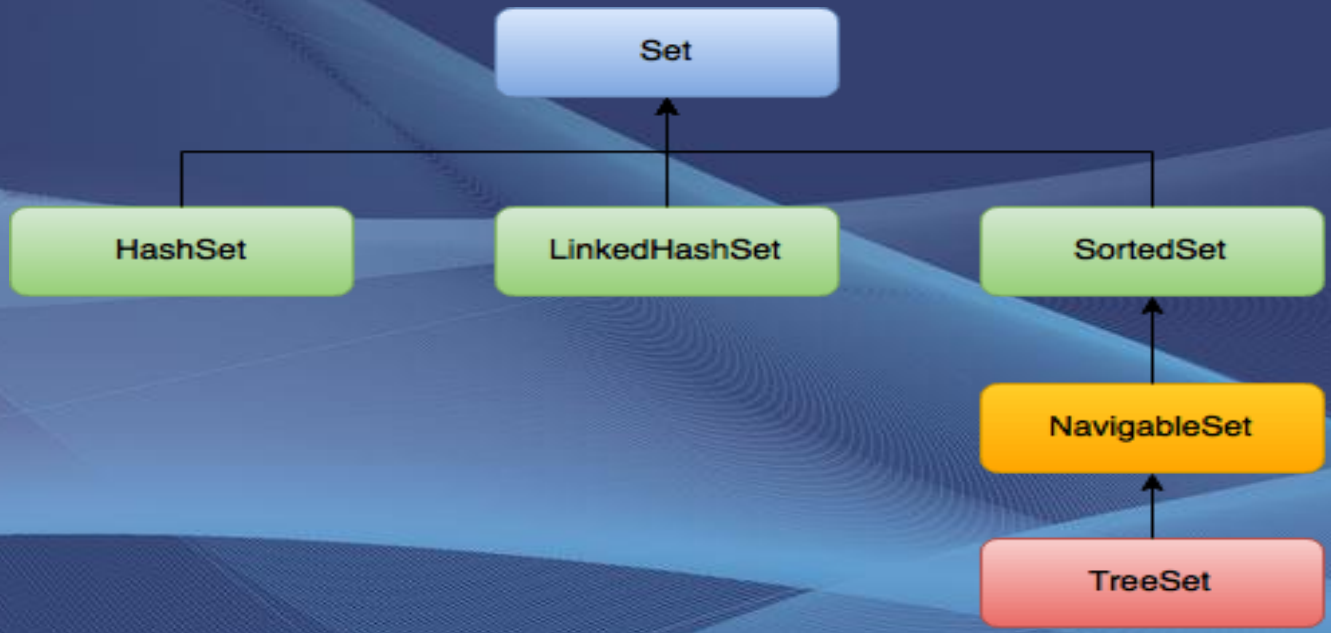
Modifier and Type	Method	Modifier and Type	Method
public Object	push(Object ele)	public Object	peek()
public Object	pop()	public int	search(Object ele)

Example for Stack:

```
import java.util.*;
Class StackDemo {
public static void main(String args[]){
Stack s = new Stack();
s.push("A");
s.push("B");
s.push("C");
System.out.println(s);  // O/P – [ A,B,C]
System.out.println(s.search("A"));  // O/P – [  3]
System.out.println(s.search("Z"));  // O/P – [ -1]
}
}
```

3) Set Interface:

- Set interface is a generic interface that extends the Collection interface.
- It provides the functionality to store and manage a set of elements and does not provide any additional methods.
- Set is useful for represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not preserved.



```
public interface Set<E> extends Collection<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); //optional
boolean remove(Object element); //optional
Iterator<E> iterator();
// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c); //optional
boolean removeAll(Collection<?> c); //optional
boolean retainAll(Collection<?> c); //optional
void clear(); //optional
// Array Operations
Object[] toArray();
<T> T[] toArray(T[] a);
}
```

HashSet:

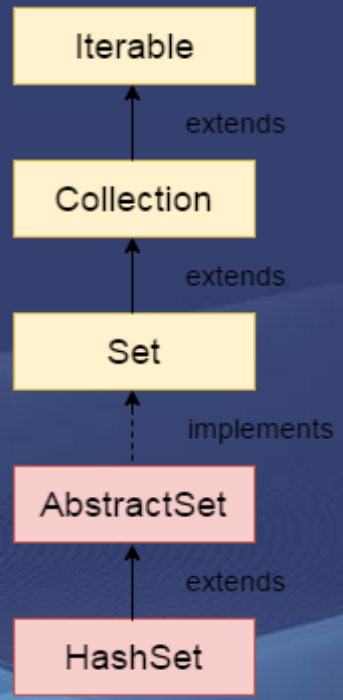
- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
  - It stores the elements by using a mechanism called hashing.
  - It contains unique elements only and 'null' insertion is possible.
  - It allows Heterogeneous objects and not allow Duplicate objects.
  - It implements Serializable and Cloneable interface but not Random Access.
  - Insertion order is not preserved and all objects will be inserted on hash-code of objects.
- Java HashSet internally uses HashMap to build Hashset (the capacity of the hash set will be 16,the load factor will be 0.75)

Declaration of HashSet:

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

Constructors of HashSet:

- HashSet()
- HashSet(Collection c)
- HashSet(int capacity)



Hierarchy of HashSet

HashSet:

```
HashSet<String> set =
        new HashSet<String>();

//Adding elements to HashSet

       set.add("RED");

       set.add("GREEN");

       set.add("BLUE");

       set.add("PINK");

//Removing "RED" from HashSet

       set.remove("RED");
```

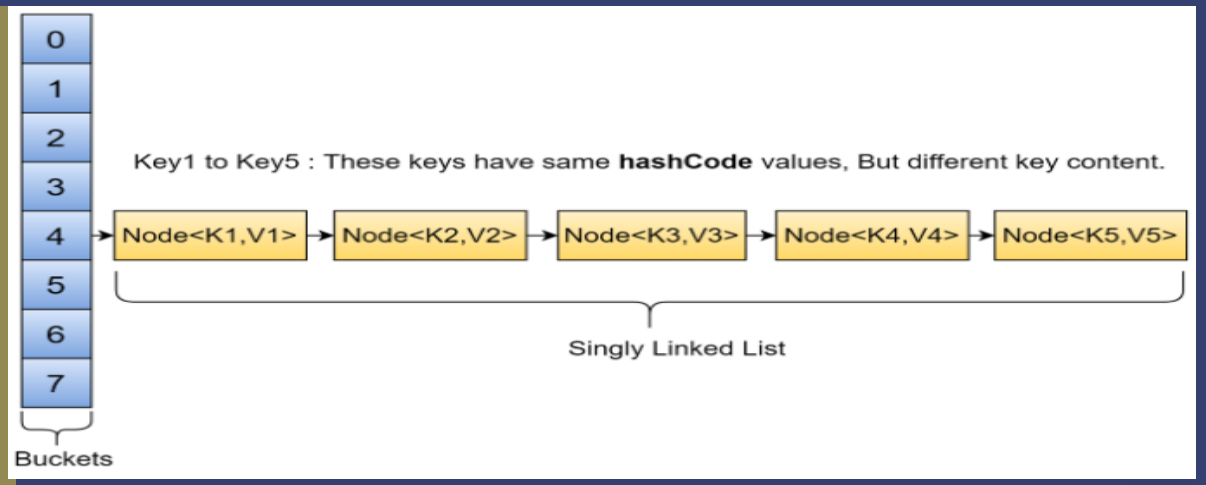
```
public HashSet()
{
  map = new HashMap<String, Object>();
}

public boolean add("RED")
{
  return map.put("RED",PRESENT)==null;
}

public boolean add("GREEN")
{
  return map.put("GREEN",PRESENT)==null;
}

public boolean add("BLUE")
{
  return map.put("BLUE",PRESENT)==null;
}

public boolean add("PINK")
{
  return map.put("PINK",PRESENT)==null;
}

public boolean remove("RED")
{
  return map.remove("RED")==PRESENT;
}
```

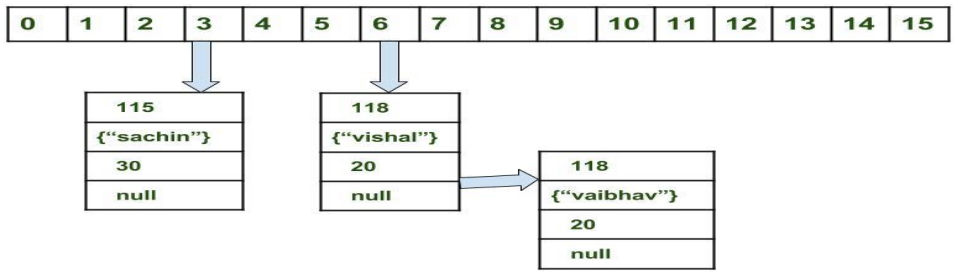
Key	Value
RED	PRESENT
GREEN	PRESENT
BLUE	PRESENT
PINK	PRESENT

Internal HashMap object

It removes an entry from HashMap object with "RED" as it's key.

Where PRESENT is a constant which is defined as private static final Object PRESENT = new Object();





Methods of HashSet:

Modifier and Type	Method	Modifier and Type	Method
public boolean	contains(Object o)	public int	size()
public boolean	add(Object o)	public void	clear()
public boolean	remove(Object o)	public Object	clone()
public boolean	isEmpty()	public Iterator	iterator()

Example for HashSet:

```
import java.util.*;
class HashSetDemo {
public static void main(String args[]){
HashSet h = new HashSet();
h.add("A");
h.add("B");
h.add(null);
h.add(10);
System.out.println(h.add("B"));  // O/P – false
System.out.println(h);  // O/P – [null, A, B, 10]
}
}
```

LinkedHashSet:

- Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.
- The important points about Java LinkedHashSet class are:
  - It contains unique elements only like HashSet.
  - It provides all optional set operations, and permits null elements.
  - It maintains insertion order.
  - It is best choice to develop cache based applications, where duplicates are not allowed and insertion order must be preserved.

- The internal data structure is HashTable and DoublyLinkedList

LinkedHashSet:

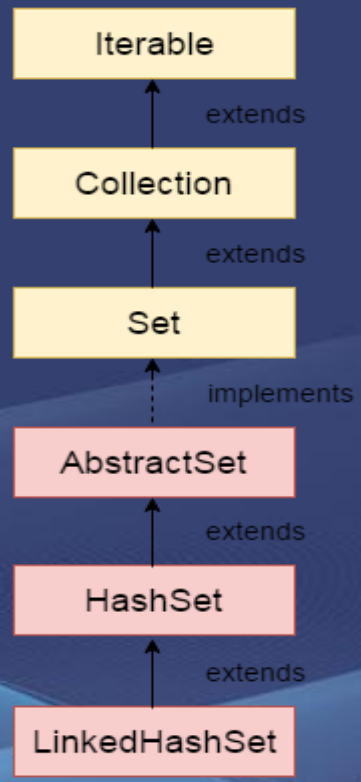
- LinkedHashMap Node(named LinkedHashMapEntry) will look like
  - Final key, value, next, before, after
  - before: points to the node inserted before this node
    after: points to the node inserted after this node
    key: key as provided
    value: value as provided
    next: points to the next node in the same bucket of array table(like in HashMap)
    hash: hashcode to calculate the index of this node, and check for equality.

Declaration of LinkedHashSet:

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

Constructors of LinkedHashSet:

- HashSet()
- HashSet(Collection c)
- LinkedHashSet(int capacity)
- LinkedHashSet(int capacity, float fillRatio)



Hierarchy of LinkedHashSet

Example for LinkedHashSet:

```
import java.util.*;
class LinkedHashSetDemo {
public static void main(String args[]){
LinkedHashSet h = new LinkedHashSet();
h.add("A");
h.add("B");
h.add(null);
h.add(10);
System.out.println(h.add("B"));  // O/P – false
System.out.println(h);  // O/P – [A, B, null, 10]
}
}
```

Sorted Set:

- It represent a group of individual objects according to some sorting order.
- It allows Homogeneous objects and not allow Duplicate objects.
- Default natural sorting order for numbers is Ascending order and for String is Alphabetic order.
- Methods of SortedSet:

Modifier and Type	Method	Modifier and Type	Method
public Object	first()	SortedSet	tailSet(Object obj)
public Object	last()	SortedSet	subSet(Object obj1, Object obj2)
SortedSet	headSet(Object obj)	Comparator	comparator()

Example for SortedSet:

```
import java.util.*;
class SortedSetDemo {
public static void main(String args[]){
SortedSet h = new TreeSet();
h.add("A");
h.add("C");
h.add("D");
System.out.println(h.add("B"));  // O/P – true
System.out.println(h);  // O/P – [A, B, C, D]
}
}
```

TreeSet:

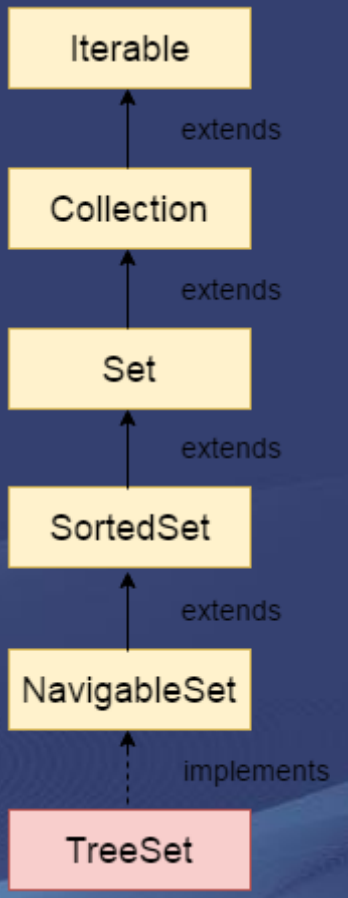
- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface.
- The objects of TreeSet class are stored in ascending order.
- The important points about Java TreeSet class are:
  - It contains unique elements only like HashSet.
  - It access and retrieval times are quiet fast.
  - It allows Null Insertion once only and not allow Duplicate objects and Heterogeneous objects.
  - Insertion order not preserved, but all objects will be inserted according to some sorting order.
  - Underlying data structure for TreeSet is Balanced Tree.

Declaration of TreeSet:

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

Constructors of TreeSet:

- TreeSet()
- TreeSet(Collection c)
- TreeSet(Comparator c)
- TreeSet(Sorted Set s)



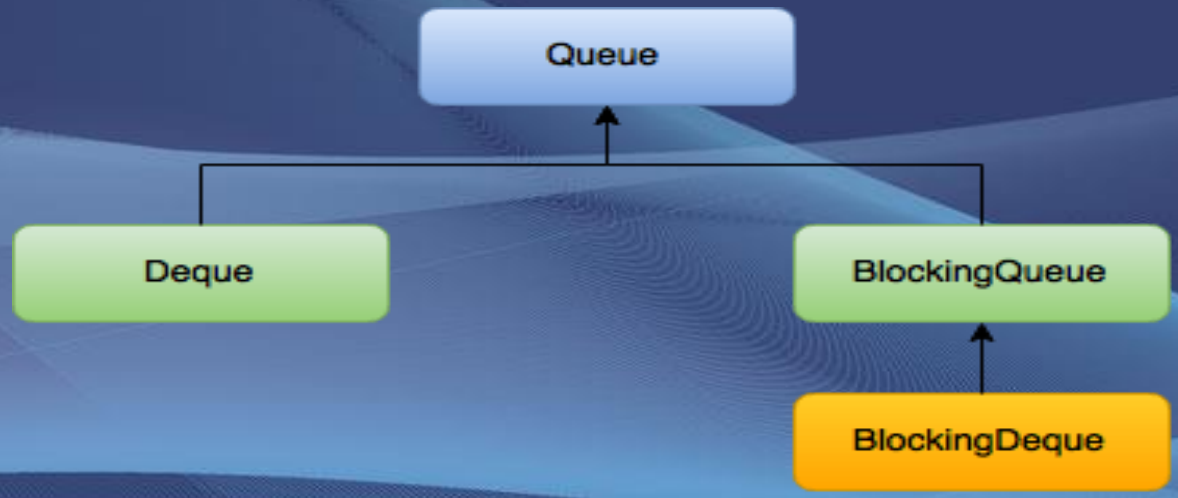
Hierarchy of TreeSet

Example for TreeSet:

```
import java.util.*;
class SortedSetDemo {
public static void main(String args[]){
TreeSet h = new TreeSet();
h.add("A");
h.add("C");
h.add("D");
System.out.println(h.add("B"));  // O/P – true
System.out.println(h);  // O/P – [A, B, C, D]
}
}
```

4) Queue Interface:

- Queue Interface extends the Collection interface to provide an implementation of a queue and provides the functionality to add, remove, access and examine queue elements.
- Typically, queues implement a first-in first out behaviour and elements can be removed only from the head.
- Queue is useful for represent a group of individual objects prior to processing.



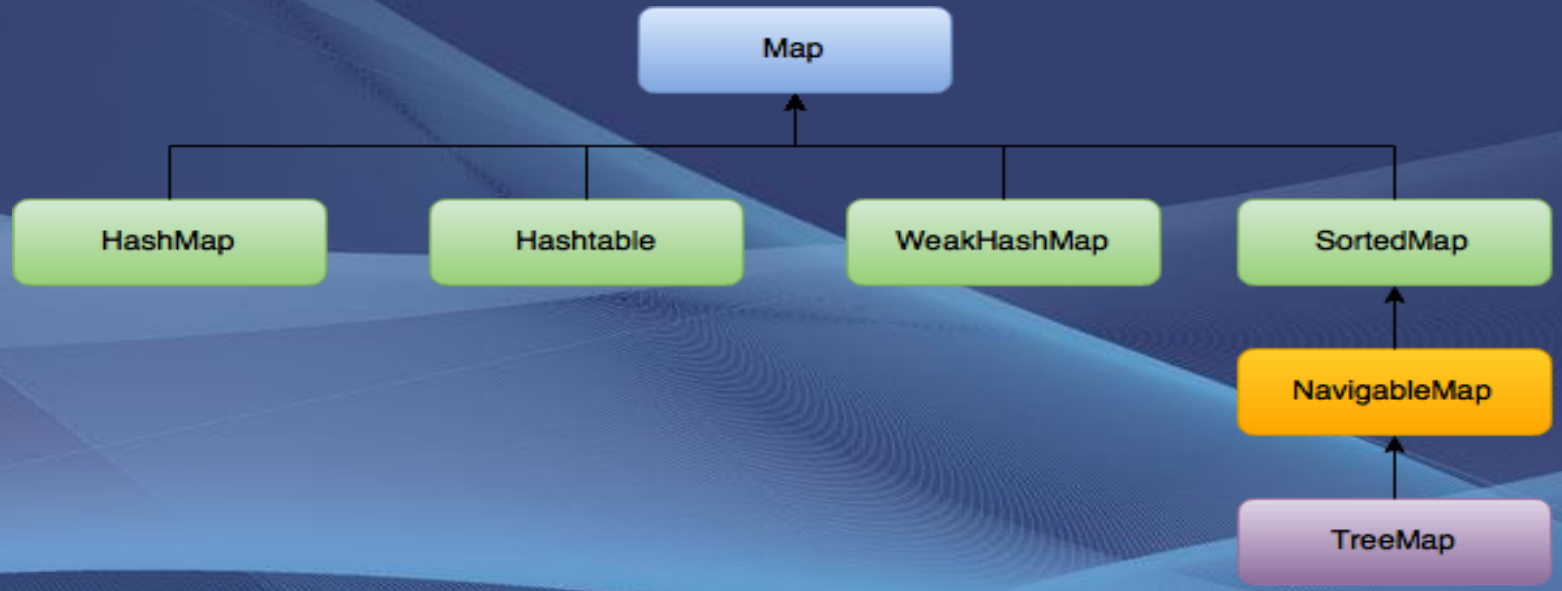
Example for Queue Interface:

```
import java.util.*;
class TestCollection {
public static void main(String args[]) {
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Monika");
queue.add("Darshan");
queue.add("Praveen");
queue.add("Suresh");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}

// Continued
```

```
queue.remove();
queue.poll();
System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
} } }
/* O/P -  head:Darshan
          head:Darshan
          iterating the queue elements:
          Darshan
          Monika
          Praveen
          Suresh
          after removing two elements:
          Praveen
          Suresh    */
```

5) Map Interface:

- The Map interface is a generic interface that provides a way to store key/value pairs. A map is an object that maps keys to values. Keys are unique and are used to identify values.
- Both key and value are objects, duplicated keys are not allowed but values can be duplicated.



5) Hash Table Interface:

- The Hash tables are used for constructing the
  - Class and its members
  - Variable lookup tables
  - Designed to provide insertion, deletion and lookup constant time

key	value
apple	120
banana	60
manago	100
orange	60

Example for Map Interface:

```
import java.util.*;
class MapInterfaceExample {
 public static void main(String args[]) {
  Map<Integer,String> map=new HashMap<Integer,String>();
          map.put(102, "Darshan");
          map.put(101, "Monika");
          map.put(103, "Praveen");
          map.put(103, "Suresh");
  for(Map.Entry m:map.entrySet()) {
System.out.println(m.getKey()+" "+m.getValue());
} }
} /* O/P -   101 Monika
                102 Darshan
                103 Suresh  */
```

HashMap	HashTable
Introduced in 1.2	legacy
not thread safe and unsynchonized	thread safe and synchronized
fast	slow
works with single thread	works with multiple thread
allow one null key	does not allow null key

Map print:

- The using iterator

```
Iterator iter = (Iterator) hasMap.entrySet().iterator();
      while(iter.hasNext())
      {
        Entry<Integer, String> info=(Entry<Integer, String>)iter.next();
           System.out.println(info.getKey()+ "::"+info.getValue() );
      }
```

- Using for loop

```
for(Map.Entry<Integer, String> map : hasMap.entrySet()) {
 System.out.println(map.getKey() + "::"+ map.getValue());
}
```

- Using Lambda

```
hasMap.forEach((K,v)->{System.out.println(K+" :: "+v);});
```

Map print:

- keyset

```
for(Integer key: hasMap.keySet()) {
      System.out.println(key + "--"+ hasMap.get(key));
 }
```

- Values

```
for(String val: hasMap.values()) {
          System.out.println(val );
     }
hasMap.entrySet().forEach(System.out::println);
hasMap.keySet().forEach(System.out::println);
 hasMap.values().forEach(System.out::println);
```

Summarization of principal classes in Java collections framework:

Principal collection class	Base class	Base interfaces	Allow duplicate elements?	Ordered?	Sorted?
ArrayList<E>	*AbstractList<E>*	*List<E>*	Yes	Yes	No
LinkedList<E>	*AbstractSequentialList<E>*	*List<E>; Deque<E>*	Yes	Yes	No
Vector<E>	*AbstractList<E>*	*List<E>*	Yes	Yes	No
HashSet<E>	*AbstractSet<E>*	*Set<E>*	No	No	No
LinkedHashSet<E>	*HashSet<E>*	*Set<E>*	No	Yes	No
TreeSet<E>	*AbstractSet<E>*	*Set<E>; NavigableSet<E>; SortedSet<E>*	No	Yes	Yes
Queue<E>	AbstratQueue<E>	Collection<E>	Yes	Yes	No

Cursors:

- Cursors are used to retrieve objects one by one from the collection.
- There are three types of cursors:

Property	1. Enumeration	2. Iterator	3. ListIterator	
Applicable for	Only legacy classes	Any Collection classes	Only List classes	
Movement	Only forward direction	Only forward direction	Both forward and backward direction	
Accessibility	Only read access	Both read and remove	Read, Remove, Replace and addition of new object	
How to get it?	By 2 methods 1. hasMoreElements() 2. nextElement()	By 3 methods 1. hasNext() 2. next() 3. remove()	By 9 methods	
			1. hasNext() 2.next() 3.nextIndex() 4.hasPrevious() 5.previous()	6. previousIndex() 7. remove() 8. set(Object new) 9. add(Object new)

Example for Enumeration Cursor:

```
import java.util.*;
public class MyEnumeration {
    public static void main(String a[]) {
        Vector<String> lang = new Vector<String>();
        Enumeration<String> en = null;
        lang.add("JAVA");
        lang.add("JSP");
        en = lang.elements();
        while(en.hasMoreElements()) {
        System.out.println(en.nextElement());
    } }
} /* O/P - JAVA
           JSP    */
```

Example for Iterator Cursor:

```
import java.util.*;
public class MapInterfaceExample
{
public static void main(String[] args)
{
   List<String> myList = new ArrayList<String>();
        myList.add("Java");
        myList.add("JSP");
        myList.add("Servlet");
System.out.println("Before remove:");
    System.out.println(myList);
          Iterator<String> itr = myList.iterator();
    while(itr.hasNext())

// Continued
```

```
{
   String removeElem = "JSP";
        if(removeElem.equals(itr.next())) {
            itr.remove();
        }
}
    System.out.println("After remove:");
    System.out.println(myList);
    }
} /* O/P -  Before remove:
              [Java, JSP, Servlet]
              After remove:
              [Java, Servlet]     */
```

Example for Iterator Cursor:

```
import java.util.*;
public class MyListIterator {
    public static void main(String a[]) {
        List<Integer> li = new ArrayList<Integer>();
        ListIterator<Integer> litr = null;
        li.add(25);
        li.add(90);
        li.add(35);
        litr=li.listIterator();
System.out.println("Forward direction");
        while(litr.hasNext()) {
System.out.println(litr.next());
        }
// Continued
```

```
System.out.println("Backward direction");
        while(litr.hasPrevious()) {
System.out.println(litr.previous());
        }
    }
} /* O/P -  Forward direction
                25
                95
                30
                Backward direction
                30
                95
                25  */
```

Generics:

- Generics are given the ability to create generalized classes, interfaces and methods by operating through references of type Object.
- It was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException.
- It provides compile-time type safety that allows programmers to catch invalid types at compile time and also expand ability to reuse code.
- Advantages:
  - Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
  - Type casting is not required : There is no need to typecast the object.
  - Compile-Time Checking : It is checked at compile time so problem will not occur at runtime.

Syntax: ArrayList<String>

Example for Generics:

```
import java.util.*;
 class TestGenerics1{
     public static void main(String args[]){
       ArrayList<String> list=new ArrayList<String>();
         list.add("rahul");
         list.add("jai");
         //list.add(32);//compile time error
       String s=list.get(1);//type casting is not required
     System.out.println("element is: "+s);
   Iterator<String> itr=list.iterator();
       while(itr.hasNext()){
     System.out.println(itr.next());
} } }
```

Annotations:

- It is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- It is an alternative option for XML and java Marker Interfaces.
- Annotations are executed by predefined tool APT(Annotation Processing Tool).
- Types of Annotations:
  1. Built-In Java Annotations
  2. Custom Annotation

Annotations:

- Two types of syntax of annotations are
  - Declaration Syntax
    - @interface annotation_name{

      datatype member [default value] ;

      ;;

      }
  - Implemeation syntax
    - Variables, members, classes..
    - @annotation_name(member1=value1,member2=value2. ..)
      Programming element

Annotations classification:

Annotations – java.lang.annotation.Annotation (super interface name)

(a) standard

1. General Purpose annotations (Built in)

@override, @deprecated, @supresswarning

@FunctionalInterface (java 8.0)

- java.lang package

2. Meta annotations

@Documented @Inherited @Target @Retention

-java.lang.annotations

(b) custom annotations

Annotations:

XML Based Tech	Annotation Based Tech
jdk1.4	jdk 1.5
jdbc 3.x	jdbc 4.0
servlet 2.5	servlet 3.0
struts 1.x	struts 2.x
JSF 1.x	JSF 2.X
Hibernate 2.4	Hibernate 3.2
Spring 2.x	spring 3.c

1.1 Built-In Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

1.2 Built-In Annotations used in other annotation

- @Target
- @Retention
- @Inherited
- @Documented

Built-In Annotations:

@Override:

It assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

@SuppressWarnings:

It is used to suppress warnings issued by the compiler.

@Deprecated:

It marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

Java Custom Annotation:

- Easy to create and use.
- The @interface element is used to declare an annotation.
- Few points for custom annotation signature:
  - Method should not have any throws clauses.
  - Method should return one of the following:

    primitive data types, String, Class, enum or array of these data types.
  - Method should not have any parameter.
  - It may assign a default value to the method.

Example: @interface MyAnnotation{}

Types of Custom Annotation:

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation: An annotation that has no method.

The @Override and @Deprecated are marker annotations.

Example: @interface MyAnnotation{}

2) Single-Value Annotation: An annotation that has one method.

@SuppressWarning("unchecked")

Example: @interface MyAnnotation{ int value(); }

3) Multi-Value Annotation: An annotation that has more than one member.

@RequestMapping(method = RequestMethod.GET, value = "/isProcess")

Example:

```
@interface MyAnnotation {
int value1();
String value2();
 }
```

Example:

```
@interface MyAnnotation {
 int value1() default 1;
 String value2() default "xyz";
 }
```

Built-in Annotations used in custom annotations in java are:
@Target, @Retention, @Inherited, @Documented

@Target: It is used to specify at which type, the annotation is used.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. The constants of ElementType enum are:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example: To specify annotation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
                int value1();
                String value2();
}
```

Example: To specify annotation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD}) @inter
face MyAnnotation{
                int value1();
                String value2();
    }
```

@Retention: It is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM. It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM.

Example: To specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

@Inherited: It is used to give meta data to other annotations. By default java doesnot allow the custom annotations to inherit.
@inherited is a type of meta-annotation used to annotate custom annotations so that the subclass can inherit those custom annotation

Example: To specify the RetentionPolicy

```
@interface MyAnnotation{
 String value default "mallik";
}

@MyAnnotation(value="I am in training");
Class A{
}

Class B extends A{

}
```

@Documented: Java annotations are not shown in the document created by Javadoc tool.To ensure our custom annotations shown in documentations we use @Documented annotations.
It is meta-annotations to annotate custom annotations.

Example: To specify the RetentionPolicy

```
@Documented @interface MyAnnotation{
 String value default "mallik";
}

@MyAnnotation(value="I am in training");
Class A{
}

Class B extends A{

}
```

Java Enum:

- It is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.
- Advantages of Java enum:
  - Improves type safety
  - Easily used in switch
  - Traversed
  - It can have fields, constructors and methods
  - It may implement many interfaces but cannot extend any class because it internally extends Enum class

- Enum has constructors. Except of constructors, an Enum is an ordinary class.
- Each name listed within an Enum is actually a call to a constructor
- Enum constructors are only available within the Enum itself

Example for Enum:

```
class EnumExample4 {
enum Season {
WINTER(5), SPRING(10), SUMMER(15), FALL(20);
private int value;
private Season(int value) {
this.value=value;
} }
public static void main(String args[]) {
for (Season s : Season.values())
System.out.print(s+" "+s.value);
} } // O/P - WINTER 5 SPRING 10 SUMMER 15 FALL 20
```

- Enum has constructors. Except of constructors, an Enum is an ordinary class.
- Each name listed within an Enum is actually a call to a constructor
- Enum constructors are only available within the Enum itself

Example for Enum:

```
class EnumExample4 {
enum Season {
WINTER(5), SPRING(10), SUMMER(15), FALL(20);
private int value;
private Season(int value) {
this.value=value;
} }
public static void main(String args[]) {
for (Season s : Season.values())
System.out.print(s+" "+s.value);
} } // O/P - WINTER 5 SPRING 10 SUMMER 15 FALL 20
```

Regular Expressions:

- The Java Regex or Regular Expression is an API to define pattern for searching or manipulating strings.
- It is widely used to define constraint on strings such as password and email validation. We will be able to test our own regular expressions by the Java Regex Tester Tool.
- Java Regex API provides 1 interface and 3 classes in java.util.regex package.
- Classes and Interface:

  java.util.regex package: It provides following classes and interface for regular expressions.

  - MatchResult (I)
  - Matcher (C)
  - Pattern (C)
  - PatternSyntaxException (C)

Matcher©:

- It implements MatchResult(I).
- It is a regex engine i.e. used to perform match operations on a character sequence.

Methods	Description
boolean matches()	test whether the regular expression matches the pattern.
boolean find()	finds the next expression that matches the pattern.
String group()	returns the matched subsequence.
int start()	returns the starting index of the matched subsequence.
int end()	returns the ending index of the matched subsequence.

Pattern(C):

- It is the compiled version of a regular expression.
- It is used to define a pattern for the regex engine.

Methods	Description
static Pattern compile(String regex)	compiles the given regex and return the instance of pattern.
Matcher matcher (CharSequence input)	creates a matcher that matches the given input with pattern.
String[] split(CharSequence input)	splits the given input string around matches of given pattern.
String pattern()	returns the regex pattern.

Character(C):

- It is used to match only one out of several characters.

Methods	Description
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Example for Regular Expression:

```
class RegexExample{
        public static void main(String args[]) {
System.out.println("by character classes and quantifiers ...");
System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949")); // O/P - true
System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949")); // O/P - true
System.out.println(Pattern.matches("[789][0-{9}", "99530389490")); // O/P - false (11 chars)
System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949")); // O/P - true
        }
    }
```

https://www.javatpoint.com/java-regex

THANK YOU