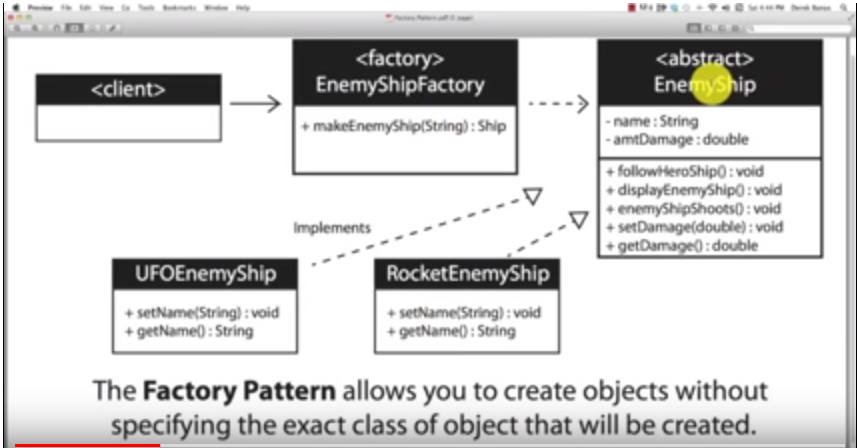
What is a design Pattern?

* a **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
* Design Patterns are targeted in solving the problems of object generation and integration.
* These are like templates that can be applied to real world problems.  
    
    
  Types of design Patterns:  
  1. **Creational Design Pattern**:   
   - Deals with object creation and initialization.  
   - Helps in deciding with objects needs to be created in a given case.  
  for example : Singleton, factory and asbtract factory.  
    
  2. **Structural Design Patterns**:  
   - Deals with class and Object composition  
   - Focuses on decoupling interface and implementation of classes and its objects.  
  for example - adapter and bridge  
    
  3. **Behavioral design Pattern:** - Deals with communication between classes are objects.  
  for example - chain of responsibility, command.

Factory Design Pattern:

Source- <http://www.newthinktank.com/2012/09/factory-design-pattern-tutorial/>

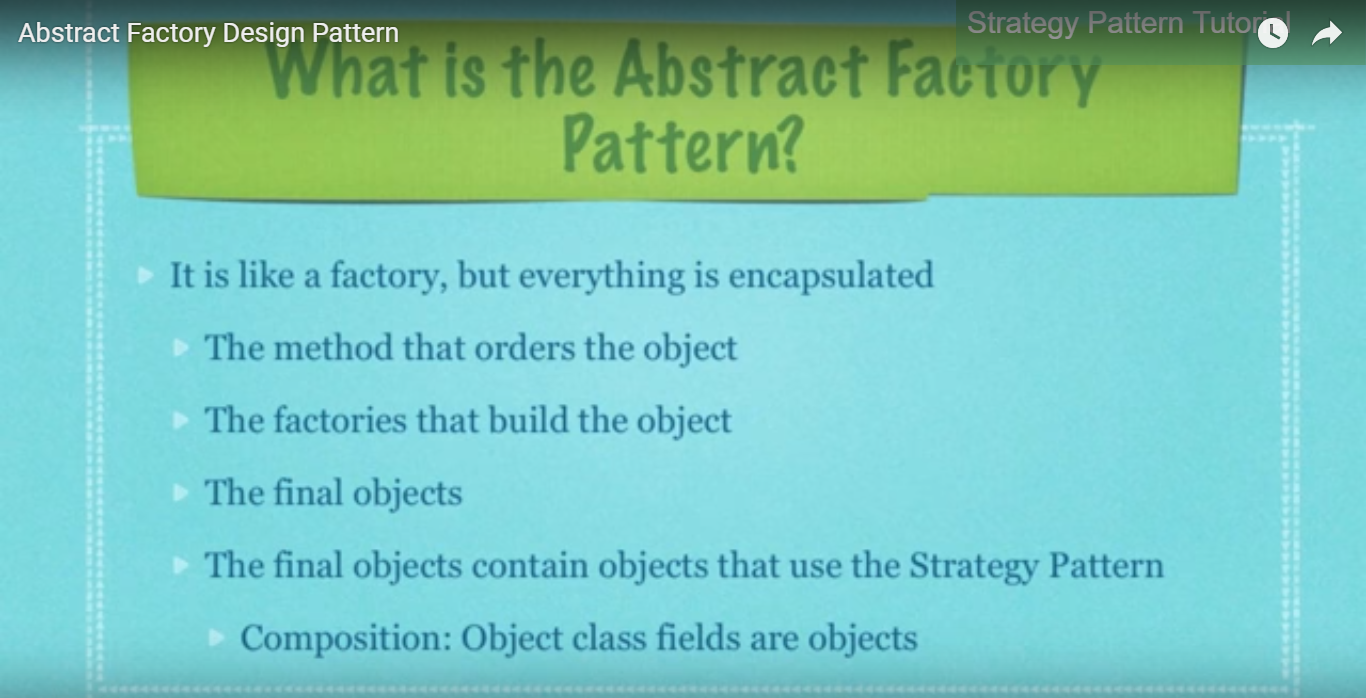
A design pattern that is used when we want to use a method to return one or more possible classes that have a common super class. That is, the class is choosen at run time.

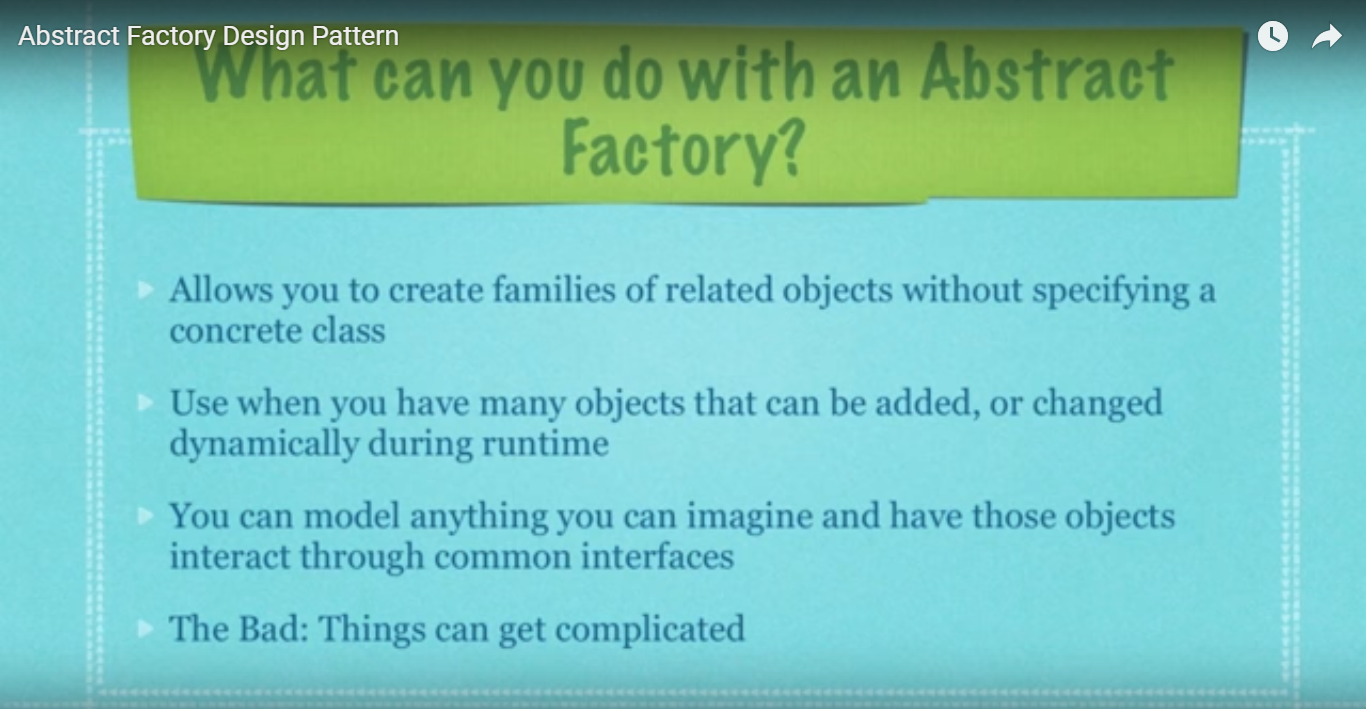


# Abstract Factory Design Pattern:

Source: http://www.newthinktank.com/2012/09/abstract-factory-design-pattern/

* Its like a factory deisng pattern but everything is encapsulated





# Singleton design Pattern:

Source:

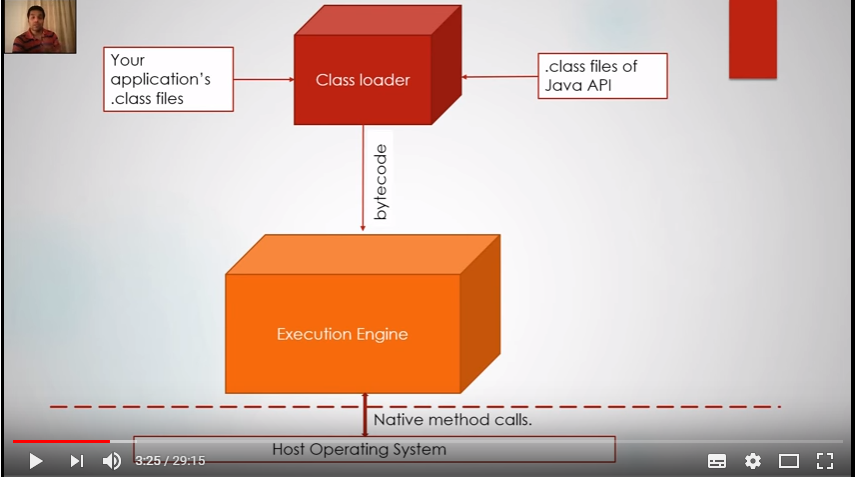
-Used when we don’t to create more than one object for a class



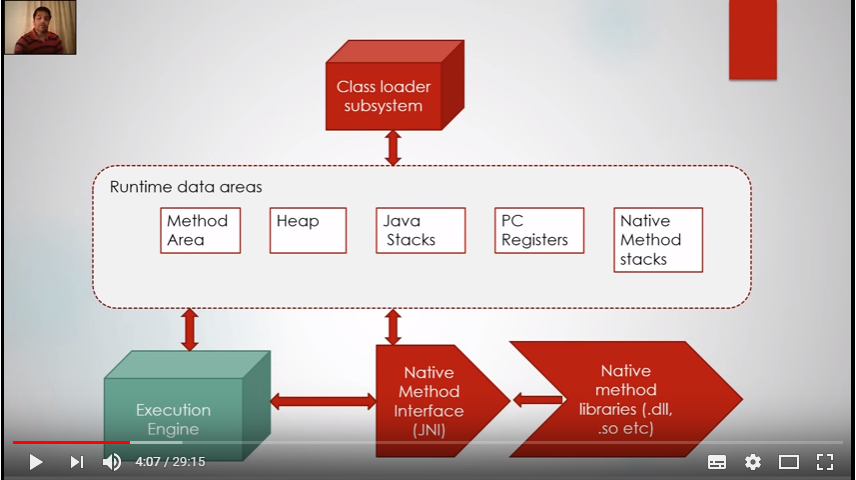
# JVM:

* Runtime environment that Loads and exeuctes the application.
* When we compile our .java file, it creates a .class file.
* JVM loads and executes this class file.

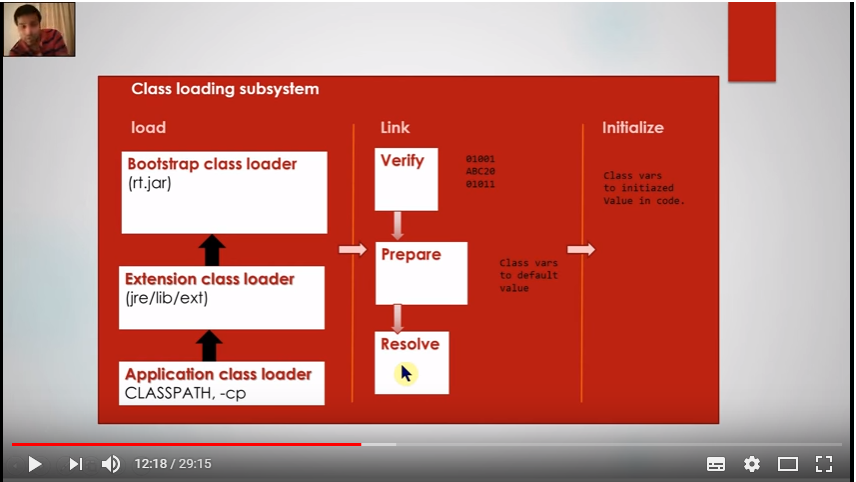
## Overall Diagram:



The class loader, loads the .class file and the inbuilt classes on Java, like string, collection classes, etc and stores them into runtime data areas.



# Class Loading Subsystem:



It has three main subsystems: Load, Link and Initialize.

1. Load:

* For Loading the bytcode from differetn sources, like, the .classfile, jar file, network socket, etc.
* Types of Loader:

1. **Bootstrap class loader** : It loads the internal classes that are present in the **rt.jar** that is distrubuted with the JRE.
2. **Extension Class Loader**: Loads additional application classes present in **JRE/lib/ext** folder
3. **Applicaiton Class Loader**: Loads classes from values specified in the classpath parameter.
4. Link: Verify,prepare and resolve
5. **Verify:** Verifies if the bytcode is valid or not. (How?)
6. **Prepare**: Memory is allocated for static (class) variable. Only memory is allocated and is set to its default value.
7. **Resolve**: All the symbolic references (like references to other classer or references to values in the constant pool) are changed into actual references.

**Class Def not found** occurs here..

1. **Initialize** Static initializers (like static blocks) are executed at this phase. Values provided to static variables are provided with their actual values set the code

Runtime Data Area:

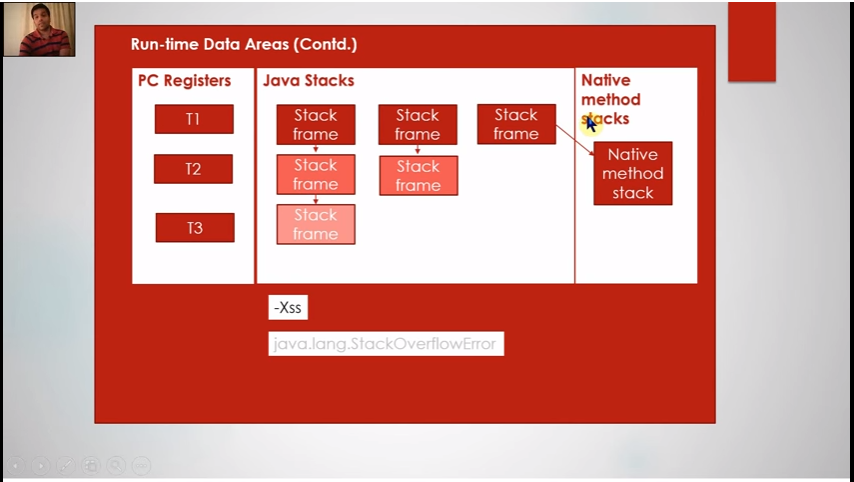


**Method Area:**

* Class data is stored in Method Area

**Heap:**

* Objects are stored in Heap.
* Its size can be adjusted using ‘-xms’ for minimum size and ‘-xmx’ for maximum size.
* Heap’s default size ¼ of the physical memory (xmx)



1. **PC Register**: Program counter register.

* Its the pointer to the next intruction to be executed in a paritcular Thread.
* This is per thread. Thread 1 cannot see whats there in the Stack frame of Thread 2. So this is Thread Safe.

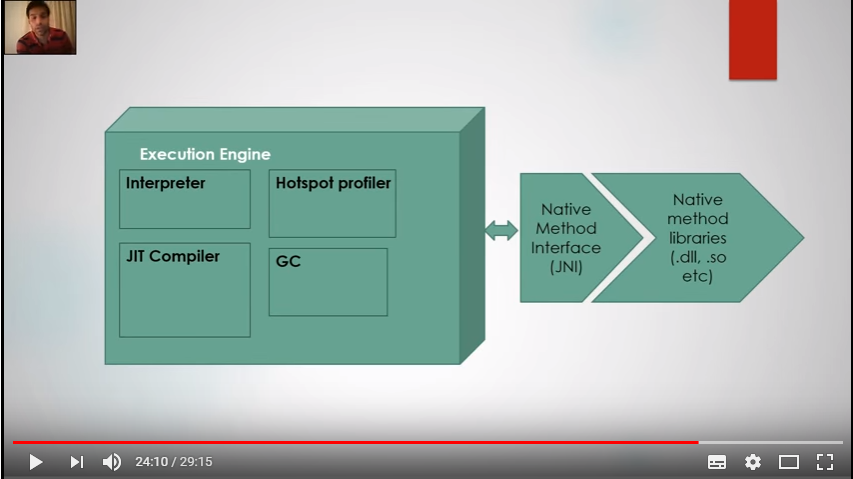
1. **Java Stacks**:

* Contains Stackframe for a particular Thread. Say, a Thread calls ‘method1’ which calls ‘method2’ and then it calls ‘method3’. Then Java stacks will have information for this stack frame (and for each thread).
* Infinite calling of the method will overload this memory anc result in stackoverflow.
* -Xss can be used to adjust the size of the Java Stacks
* This is per thread. Thread 1 cannot see whats there in the Stack frame of Thread 2. So this is Thread Safe.

1. **Native Stack Methods:**

* Used for storign Navtive methods.
* This is per thread. Thread 1 cannot see whats there in the Stack frame of Thread 2. So this is Thread Safe.

# Execution Engine:



Interperter:

* Interpretes the current instruction in the bytecode and executes it.
* Interpreter checks the Native library to exeucte the instructions.

JIT Compiler:

* It helps to skip the interpretions of the instructions that are exeucted multiple times to save the time spent in interpretation.

Hotspot Profiler:

* Used to keep track some statics related to the bytecode like which instrucitons are runing multiple times, etc.

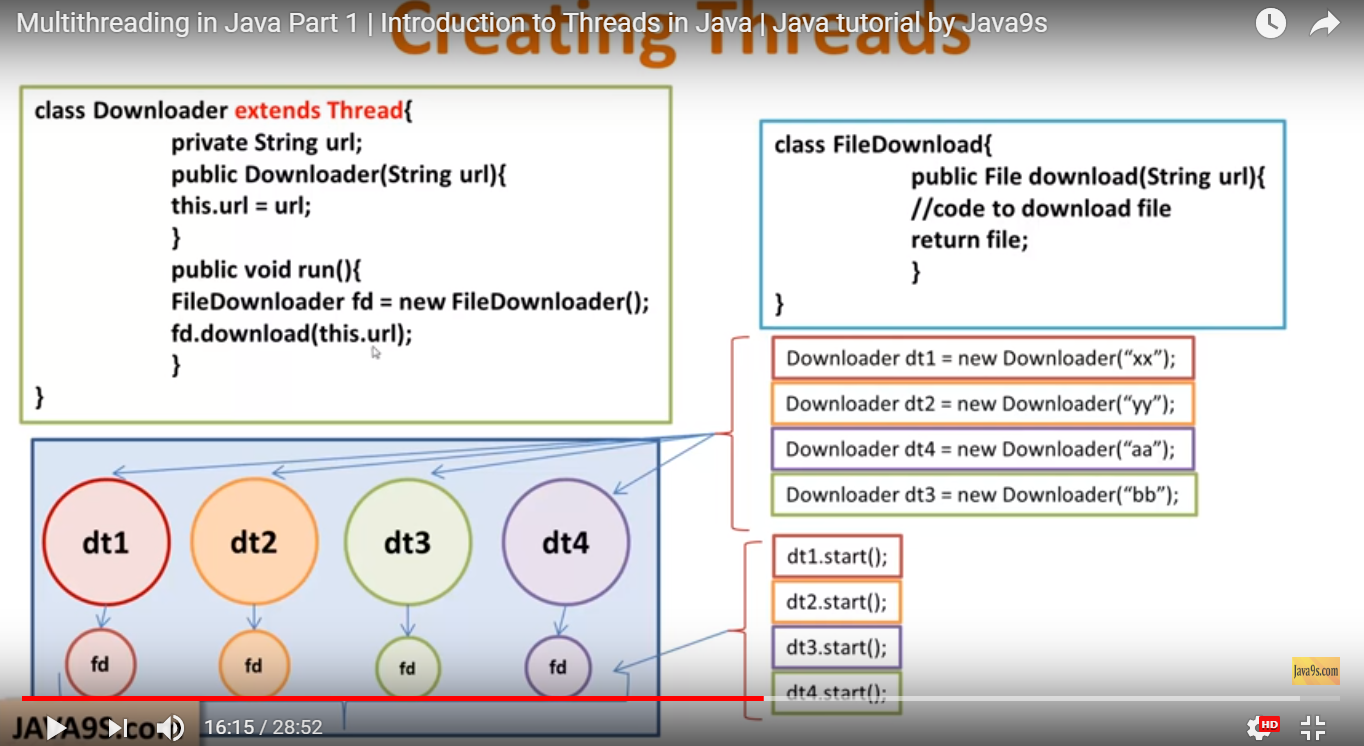
# Multi Threading:

* A Thread is :

1. A Thread of execution
2. Instance of Java .lang.Thread.

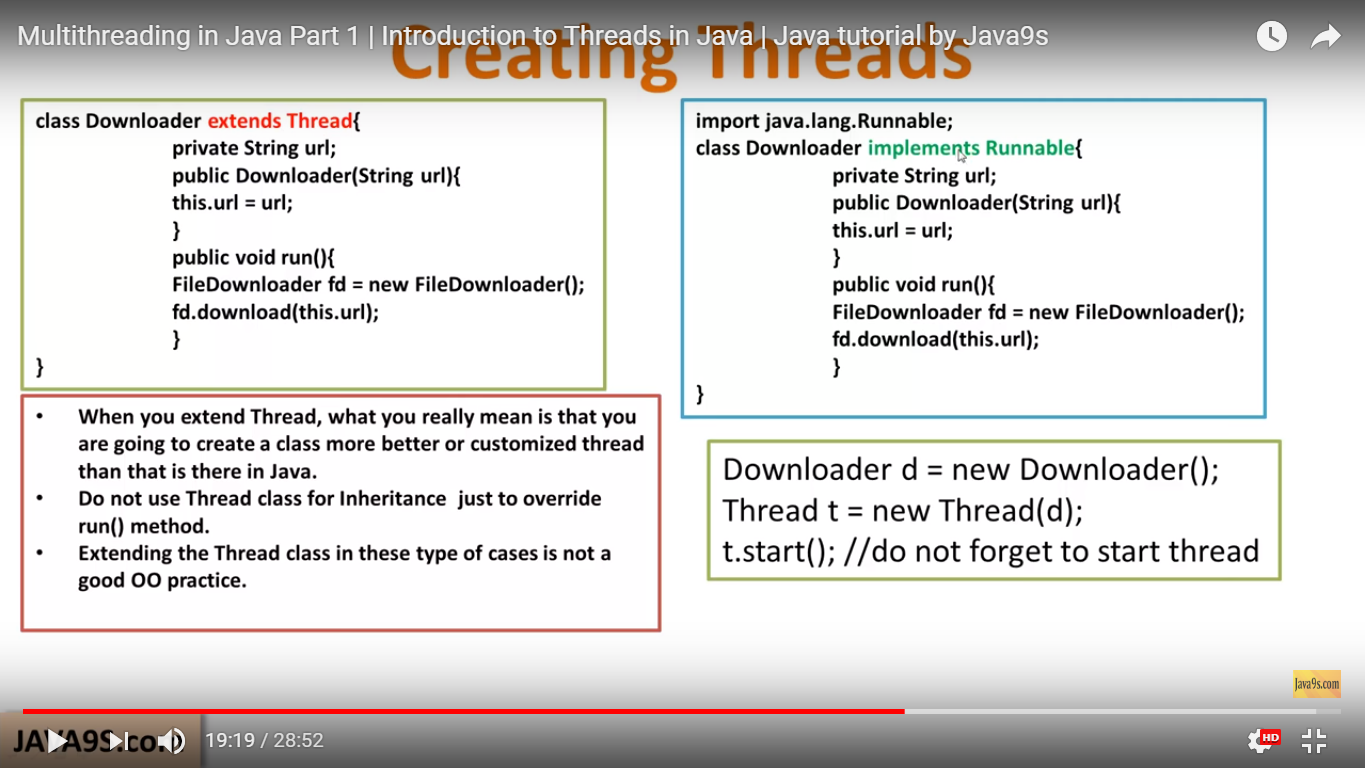
We can Create Thread in two ways:

1. Extending the Thread Class:

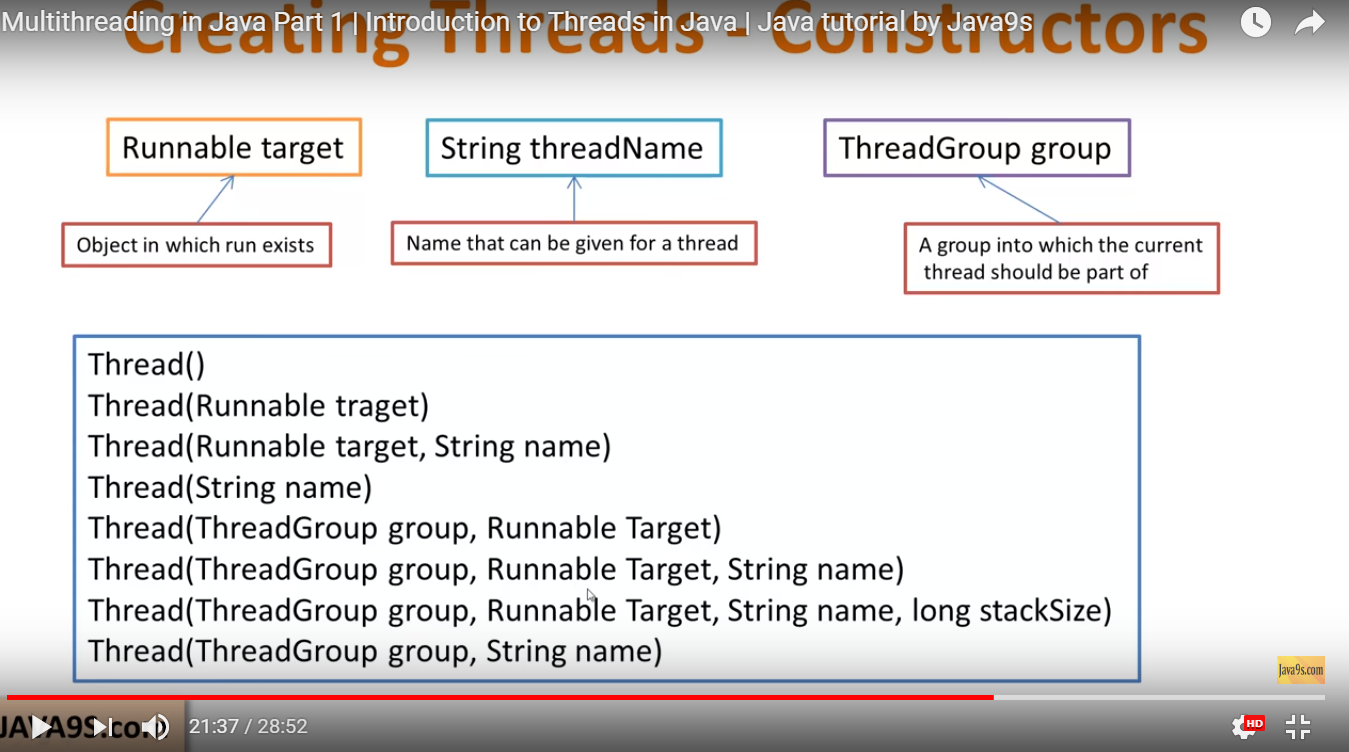


2. Implementing the Runnable Interface:

This is more preferable way as it allows the class to extend other classes as well.



Differnet type of Constructors Thread class has:



# Object Oriented Programing:

Important:

* A variable passed in as an argument is passed by value. An object passed in as a argument is passed by reference.
* Abstract classses are similar to having polymorphism
* With polymorphism we can’t access methods that are only in subclasses and not in parent class. For this we can use abstract class or interface.

When to use Abstract and when to use Interface?

1. Use interface when non of the subclasses have any common functionality for the method that they will be implementing.
2. Use interface when you want to achive multiple inheritance.
3. When we are sure that functionality won’t change in the future than we use Interface. Suppose we want to add a new method in the interface, then we need to implement that in all the subclasses. This is not the case with abstract class, we can create a new method with empty body.

Important Questions?

1. Difference between equals() and == in Java?

* ‘==’ is used to compare primitive types and objects. If we are comparing objects then it will check if both the objects are refereing to the same memory location or not.

String obj1 = new String(‘abc’);

String obj2 = new String (‘abc’);

String obj3 = obj1

Syso(obj1 == obj2 ) -🡪 False as both refer to different memory location.

Syso(obj3 == obj2) -🡪 True as both refer to the same memory location.

* **by default equals() will behave the same as the “==” operator and compare object locations. But, when overriding the equals() method, you should compare the values of the object instead.**

## Equals Method:

### public boolean equals(Object obj)

* This method checks if some other object passed to it as an argument is equal to the object on which this method is invoked.
* The default implementation of this method in Object class simply checks if two object references x and y refer to the same object. i.e. It checks if x == y.
* This particular comparison is also known as "shallow comparison".
* However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"; by actually comparing the relevant data members.
* Since Object class has no data members that define its state, it simply performs shallow comparison.