Chapter 1: Introduction to Design Patterns→

What is a design pattern?

Pattern is a solution to a problem in a context.

In general, a pattern has **four** essential elements:
The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two

The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is oft en a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

Design patterns are classified into 3 types based on **purpose** of the pattern, reflects what a pattern does.

1. **Creational patterns**: Concern the process of object creation
   a. Factory method
   b. Abstract Factory
   c. Builder
   d. Prototype
   e. Singleton

2. **Structural Patterns:** Deal with the composition of classes or objects
   a. Adaptor
   b. Bridge
   c. Composite
   d. Decorator

f. Façade
g. Flyweight
i. Proxy

3.**Behavioral patterns**: Characterize the ways in which classes or objects
Interact and distribute responsibility.
a. Chain of responsibility
b. Command
c. Iterator
d. Interpreter
e. State
f. Strategy
g. Memento
h. Mediator
i. Observer
j. Template
k. Visitor pattern

Design patterns are classified into 2 types based on **SCOPE** , specifies whether the pattern applies primarily to **classes** or to **objects**. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.

Class scope patterns:
a. Factory Method
b. Adapter
c. Interpreter
d. Template Method

Object scope patterns:

Creational Category:
a. Abstract Factory
b. Builder
c. Prototype
d. Singleton

Structural Category:
e. Adapter
f. Bridge
g. Composite
h. Decorator
i. Facade
j. Flyweight
k. Proxy

Behavioral Category:
l. Chain of Responsibility
m. Command
n. Iterator
o. Mediator
p. Memento
q. Observer
r. State
s. Strategy
t. Visitor

Brief description about Patterns:

Creational Patterns:
This design patterns is all about class instantiation, object creation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Factory method pattern:
It is class scope pattern; it creates an instance of several derived classes
Abstract Factory pattern: Creates an instance of several families of classes,
Builder pattern: Separates object construction from its representation,
Prototype pattern: A fully initialized instance to be copied or cloned,
Singleton Pattern: A class of which only a single instance can exist.

Structural Patterns:
This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Adaptor pattern: It is class scope pattern, Match interfaces of different classes.
Bridge Pattern: Separates an object's interface from its implementation
Composite pattern: A tree structure of simple and composite objects.
Decorator Pattern: Add responsibilities to objects dynamically
Façade Pattern: A single class that represents an entire subsystem
Flyweight pattern: A fine-grained instance used for efficient sharing.
Proxy Pattern: An object representing another object.
Behavioral Patterns: This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Chain of responsibility pattern: A way of passing a request between a chain of objects.
Command pattern: Encapsulate a command request as an object.
Iterator pattern: Sequentially access the elements of a collection.
Interpreter pattern: It is class scope pattern, A way to include language elements in a program.
State Pattern: Alter an object's behavior when its state changes
Strategy pattern: Encapsulates an algorithm inside a class
Memento Pattern: Capture and restore an object's internal state
Mediator Pattern: Defines simplified communication between classes
Template Pattern: It is class scope pattern, Defer the exact steps of an algorithm to a subclass
Observer pattern: A way of notifying change to a number of classes
Visitor Pattern: Defines a new operation to a class without change.

Chapter 2: Threading and Synchronization→

What is Process?

A program under exaction is called process, Process is a runtime/execution time entity, but program is a design time entity.
-Process provides the resources needed to execute a program
A Process has a self-contained execution environment;
Owns: Stack, Program Counter, variables, Memory, Filehandles, Process States
Each process has one Main thread.
System threads – GC, Object finalization, JVM housekeeping

What is a thread?

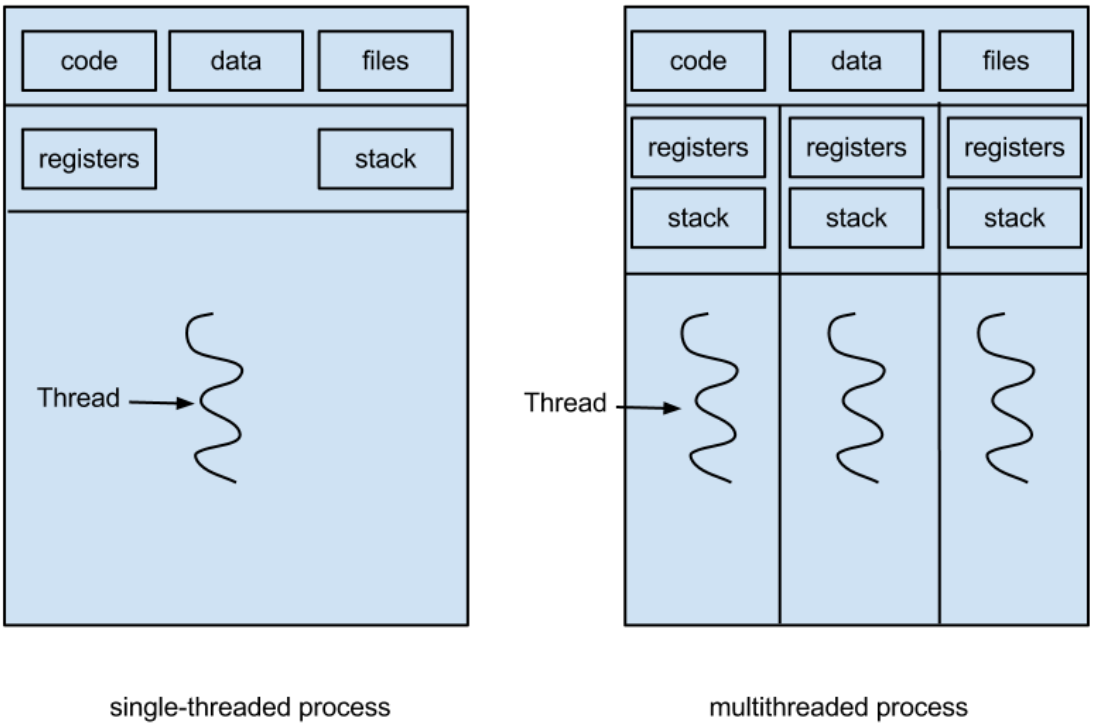
Thread is a lightweight process, some part of a process. Threads are called lightweight processes. Multiple threads are executed within a process.
Threads allow multiple program flows to coexist within the process.
Even though threads share process-wide resources (memory, file handlers), but each thread has its own Program Counter, Stack and local variables
It shares process's virtual address space and system resource.
Threads are spawned from Processes. So all the threads created by one particular process can share the process's resources (memory, file handlers, etc).

Thread Owns: Stack, Program Counter, Local variables
Thread Shares: Memory, Filehandles, Process States
ThreadGroups :Grouping threads into a logical collection; Not used much.
ThreadPools :A thread pool is a collection of threads set aside for a specific task;
Ex: webserver thread pool; saves thread creation overheads every time;

- execute(Runnable command)
- Used for executing large numbers of asynchronous tasks
- provide a boundary mechanism to create and managing the resources within the pool
- Better thread management; Cleaner shutdown of threads;
- Ability to Add, Remove, Enumerate future tasks; Apt for a scheduler;

Multithreading: Executing/Running two or more threads at a time is called multithreading.
Executing/Running two or more threads within a process is called multithreaded process.

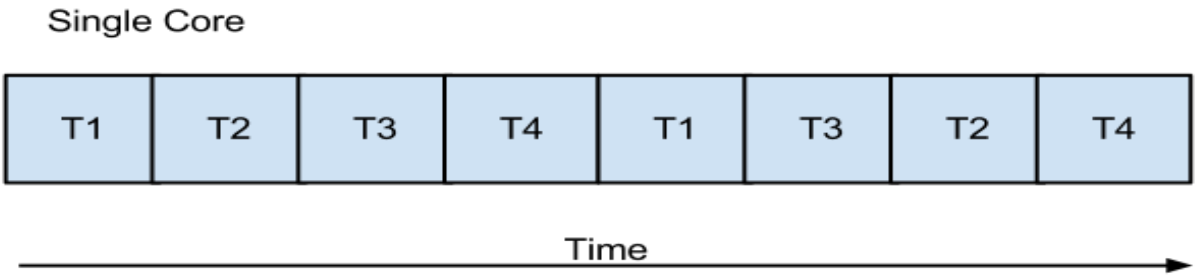
Thread and Process:

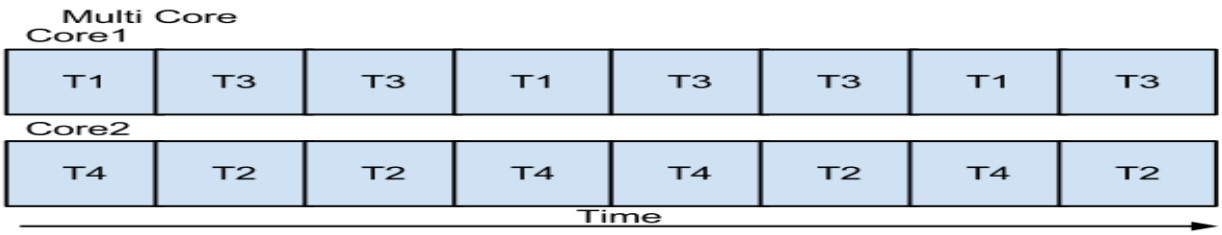


single-threaded process

multithreaded process

Benefits of Thread:





Multitasking:

A multithreaded program contains two or more parts that can run concurrently.

Each part of such a program is called a *thread, and each thread defines* a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Two Types of Multitasking:

1) Process Based
2) Thread Based

Process Based Multitasking:

a. Two or more than two programs execute concurrently. For eg. Typing in MS-WORD and listening a song.

b. Running Heavyweight tasks. Eg. Run the Java compiler at the same time that you are using a text editor.

c. It has separate address space

Inter-process communication is expensive and limited Where as Inter-thread communication is inexpensive. Switching between one thread to another is easy.

Thread based Multitasking:

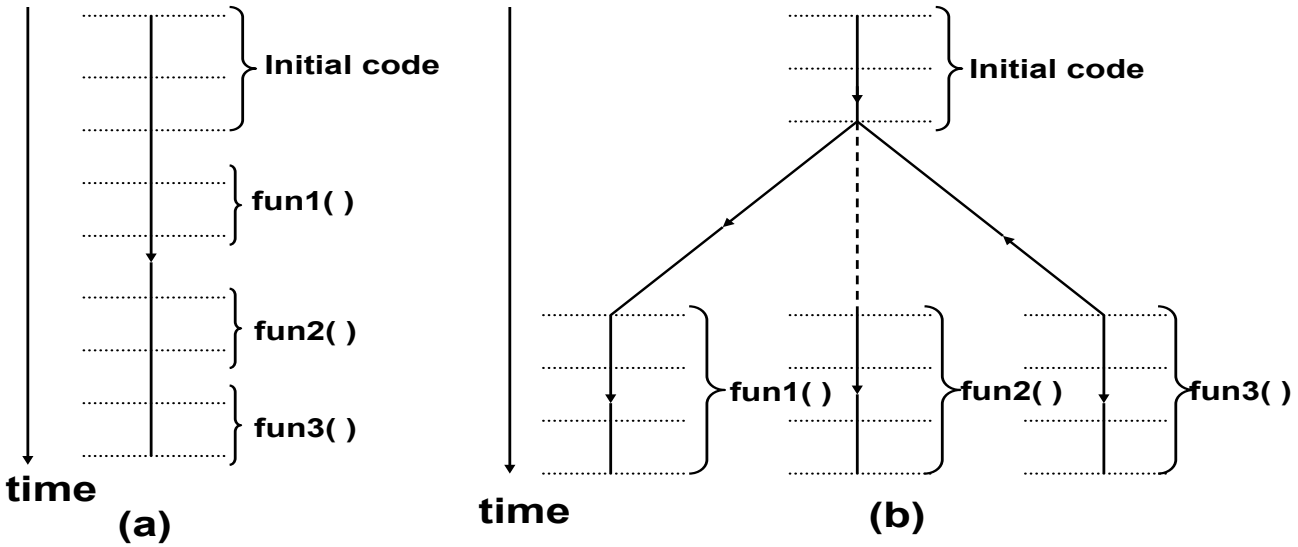
a. Single program perform two or more threads simultaneously. For eg. we will saving the file and order to print the file.

b. Running the Lightweight tasks. Eg: text editor can format text at the same time that it is printing

c. Thread Based Multitasking shares the address space.

Multithreading:

A Conceptual Picture



(a) singlethread program
(b) multithreaded program

Risks with Multithreading:

Safety Hazards
Synchronization
Liveness Hazards
Deadlock: A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
Example
System has 2 tape drives
P_1 and P_2 each hold one tape drive and each needs another one
Starvation:A process is perpetually denied necessary resources to process its work.
Starvation is a permanent blocking of one or more threads in a multithreaded application.
Starvation may be caused by errors in a scheduling or mutual exclusion algorithm, but also caused by resource leaks, and intentionally caused via a denial-of-service attack.
Race condition:A race condition occurs when 2 or more threads access shared data and they try to change it at the same time.
Performance Hazards
context switch
synchronization

Thread safety:
Behaves correctly when accessed from multiple threads.
– java.text.SimpleDateFormat is not thread safe
– Stateless servlet is safe
Race conditions:
The output is dependent on the sequence or timing of other uncontrollable events
1)

```
if(!map.containsKey(key)){
map.put(key,value);
}
```

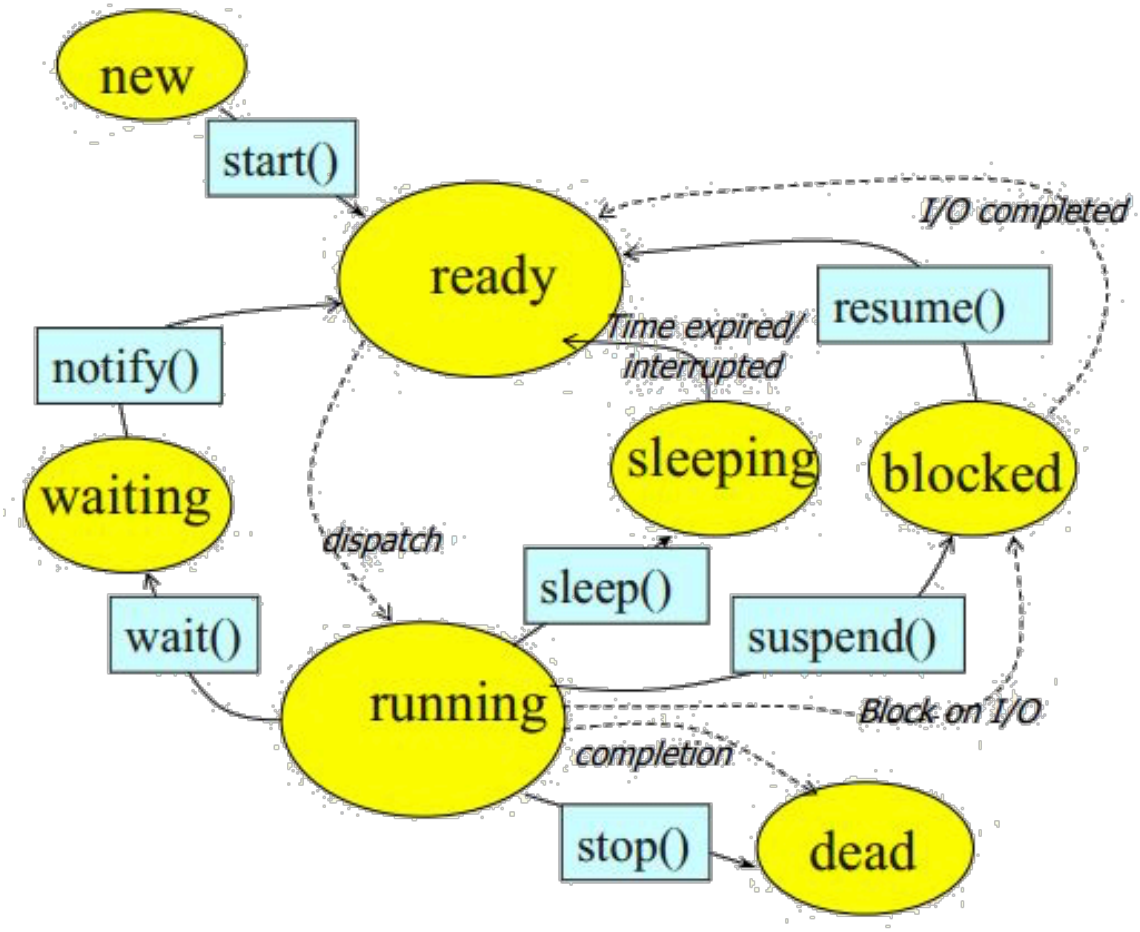
2)

```
int n;
int calculate(){
return n++;
}
```

Synchronized:
● Only one thread can execute the block of code protected by the same lock at the same time
● Ensures that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock

```
synchronized(object) {
//do something...
…
}
```

Thread Life Cycle/Thread State Diagram:



Creating Threads in Application/Program:
In Java, Threads can be created in TWO ways.
1.Subclassing the **Thread** class and instantiating a new object of that class.
2.Implementing Runnable Interface.
In both cases the **run()** method should be implemented

There is a class called Thread in java.lang package and you can create a thread of your own by deriving a class from java.lang.Thread class. This is how it is done.

```
class ExampleThreadClass extends Thread
    {
     public void run( )
```

```
        {
        - - - - - - -
        // do something useful
        - - - - - - -
        }
}
```

To call the run() method, you have to create an instance of the class ExampleThreadClass and invoke the start() method, as shown below :

```
ExampleThreadClass  thread = new  ExampleThreadClass();
thread.start();
```

This may look bit confusing since you have to start the method run() by calling a method start() which is not there in the class ExampleThreadClass.
You have created ExampleThreadClass by extending the Thread class which look something as shown below :

```
class Thread
   {
    public void start()
        {
         - - - - - -
         - - - - - -
        }
    --------------
    --------------
   }
```

thread.start() is equivalent to calling Thread.start() which will cause thread.run() running.

Declaring an object of class ExampleThreadClass creates a new thread .
The run() method is the place where we specify what a thread has to perform. The run method is invoked by calling the start() method.

You can derive two subclasses from the base class Thread with two different run() methods. Declaring two thread objects and invoking the start() method with each thread object will give you two independently executing threads of control.

Thread.start() - calls the run() method
Thread.stop() - stops the thread
Thread.suspend() - suspends a thread execution
Thread.resume() - resumes a suspended thread
Thread.sleep(100) - sleeps for 100 milliseconds
Thread.wait() - waits for something to happen
Thread.yeild() - 1.Causes the currently executing thread object to temporarily pause and allow other threads to execute,
2. Allow only threads of the same priority to run.
3. Yields the thread control to another thread
Thread.setPriority() - set thread priority of the current thread.

Thread.getPriority() - gets thread priority of the current thread.
Thread.currentThread() - gets the thread in which the method is running.
Thread.getName() - gets the name of the thread in which the method is running.
Interrupt() - Interrupts the wait state of the thread; invoked by the thread owner;
Raises a InterruptedException
Join() -- Makes the calling thread wait until other thread completes;
Typical usage: make sure all the child threads are terminated;
Can be interrupted by the thread.interrupt call
Notify() – Wakes up the thread waiting on the given object's monitor
notifyAll() – Wakes up all the threads waiting on the given object's monitor

An Example Multithreaded Program:

```
class Example1Thread1Class extends Thread
    {
     public void  run()
          {
           for(int i=0; i<10;i++)
                {
                System.out.println("Hello Thread1");
                }
          }
    }
class Example1Thread2Class extends Thread
     {
     public void run()
          {
          for(int i=0; i<10;i++)
               {
               System.out.println("Hello Thread2");
              }
         }
   }

public class Example1
    {
    public static void main(String argv[] )
          {
          Example1Thread1Class thread1= new    Example1Thread1Class();
          Example1Thread2Class thread2= new Example1Thread2Class();
          thread1.start();
          thread2.start();
          }
    }
```

The above program created the following output in one run.

```
Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1
Hello Thread1
Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1
Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1
```

The job assigned to the first thread is to print 10 lines of *Hello Thread1* message. The job assigned to the second thread is to print 10 lines of *Hello Thread2* message. The CPU is time sharing between the two threads. CPU will execute the run() method of thread1 for a time slice and prints a few lines of *Hello Thread1* message. CPU control will go to the run() method of thread2 for the next time slice and prints a few lines of *Hello Thread2* message. This process goes on.

Another Example:

```
class SimpleThread extends Thread
    {
    public SimpleThread(String str)
        {
         super(str);
         }
  public void run()
      {
      for(int i=0;i<10;i++)
          {
          System.out.println(getName());
         }
      }
  }

public class Example2
    {
    public static void main(String argv[])
        {
        new SimpleThread("Thread1").start();
```

```
            new SimpleThread("Thread2").start();
        }
    }
```

The above program created the following output in one run

```
Thread1
Thread1
Thread1
Thread1
Thread1
Thread1
Thread1
Thread2
Thread2
Thread2
Thread2
Thread2
Thread1
Thread1
Thread2
Thread2
Thread2
Thread2
Thread2
Thread1
```

In this example we have only one run() method. Two threads are sharing the same run method. The class method getName() is called to get the thread name in which the run() method is currently running. The class SimpleThread is a direct descendent of the class Thread and hence the method getName() of Thread class could be called.

Creating Treads using Runnable Interface:

The Runnable interface looks like this:

```
public interface Runnable
    {
     public abstract run();
    }
```

Creating Threads Using The Runnable Interface:

Threads can be created by implementing the Runnable interface. Following steps are involved.

Create a class by implementing Runnable interface.

```
 1.public class ExampleThreadClass implements Runnable
        {
         public void run()
              {
               ------------
               ------------
              }
        }
```

2. Create an instance of that class

ExampleThreadClass aETC = new ExampleThreadClass( );

3. Pass that instance to the constructor making a new thread

Thread aThread = new Thread (aETC);

4. Then the statement

aThread.start( ); //calls the run() method indirectly

Implementing Runnable Interface:

An Example Program:

```
class Example3ThreadClass implements Runnable
    {
     public void run()
          {
           for(int i=0;i<10;i++)
               {
                System.out.println(Thread.currentThread().getName());
               }
          }
    }
public class Example3
    {
     public static void main(String argv[])
          {
           Example3ThreadClass aETC= new Example3ThreadClass();
          Thread thread1=new Thread(aETC," My Thread");
          Thread thread2=new Thread(aETC, "Your Thread");
          thread1.start( );
          thread2.start( );
         }
  }
```

The above program generated an output as shown below in one run.

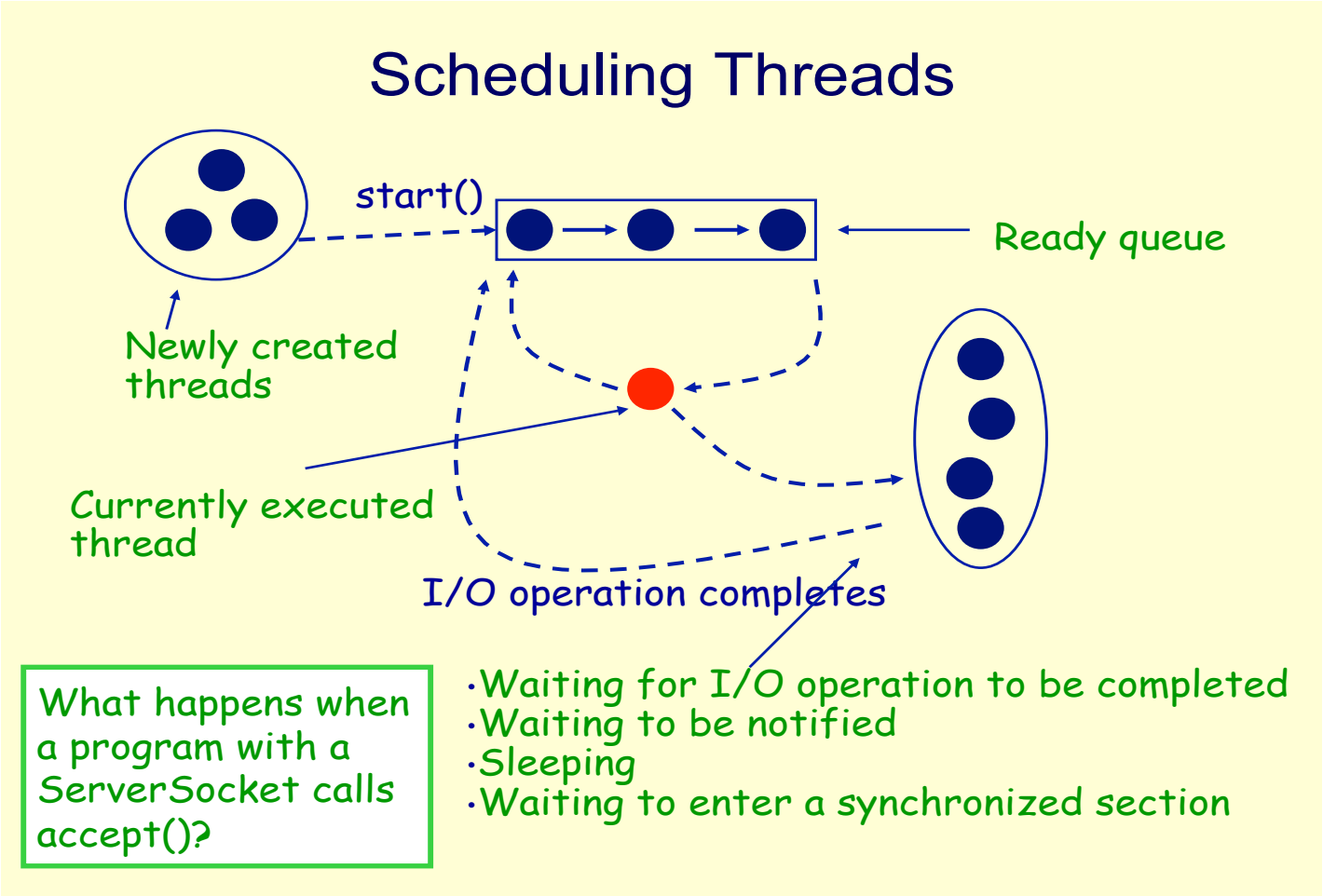
```
My Thread
Your Thread
Your Thread
Your Thread
Your Thread
My Thread
My Thread
My Thread
My Thread
Your Thread
Your Thread
Your Thread
Your Thread
My Thread
My Thread
My Thread
My Thread
My Thread
Your Thread
Your Thread
```

Thread Scheduling:
Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time.

1. Pre-emptive Scheduling
2. Non Pre-emptive Scheduling

Non Pre-emptive Systems: In non pre-emptive environments the programmer has to take steps to ensure that different threads are getting executed concurrently. *yield()* and *sleep()* methods can be used.
In non pre-emptive scheduling, the scheduler never interrupts a running thread
The non pre-emptive scheduler relies on the running thread to yield control of the CPU so that other threads may execute

Pre-emptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute



A Java program which makes sure that both the threads will be running irrespective of the scheduling strategy. *yield()* method in use.

```
class Example5ThreadClass implements Runnable
      {
       public void run()
            {
             while(true)
                  {
                      System.out.println(Thread.currentThread().getName());
                      Thread.yield();
                  }
            }
      }
public class Example5
   {
   public static void main(String argv[])
      {
      Example5ThreadClass aETC=new Example5ThreadClass();
      new Thread(aETC,"boys").start();
      new Thread(aETC,"girls").start();
      }
```

The above program printed the following output in one run.

```
       :
       :
    girls
    girls
    boys
    boys
    girls
    boys
    girls
    boys
    girls
    boys
    girls
     :
     :
```

until the program was interrupted.

A Java program which makes sure that both the threads will be running alternatively irrespective of the scheduling strategy. *sleep()* method in use.

```
class Example6ThreadClass implements Runnable
   {
   public void run()
```

```
        {
        while(true)
            {
            System.out.println(Thread.currentThread().getName());
            try
              {
              Thread.sleep(100);
              } catch (InterruptedException ie)
                    {
                     return;
                    }
              }
          }
      }


public class Example6
    {
    public static void main(String argv[])
        {
        Example6ThreadClass aETC=new Example6ThreadClass();
        new Thread(aETC,"boys").start();
        new Thread(aETC,"girls").start();
        }
  }
```

The above program printed an output as shown below:

```
boys
girls
boys
girls
boys
girls
boys
  :
  :
```

until the program was interrupted.

Thread Priorities:

Threads have priorities that can be set and changed. A higher priority thread executes ahead of a low priority thread. Threads take off with the same priority**.**

Every thread has a priority.

When a thread is created, it inherits the priority of the thread that created it.

The priority values range from 1 to 10, in increasing priority.

The priority can be adjusted subsequently using the **setPriority()** method.

The priority of a thread may be obtained using **getPriority().**

Priority constants are defined:

-MIN_PRIORITY=1

-MAX_PRIORITY=10

-NORM_PRIORITY=5

Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
The Java VM will not exit if non-Daemon threads are executing
The Java VM will exit if only Daemon threads are executing
Daemon threads die when the Java VM exits

A Java program to illustrate how priorities work:

```
class Example7ThreadClass implements Runnable
   {
   public void run()
      {
      while(true)
         {
           System.out.println(Thread.currentThread().getName());
         }
      }
   }


public class Example7
   {
   public static void main(String argv[])
      {
      Example7ThreadClass aETC= new Example7ThreadClass();
      Thread thread1=new Thread(aETC,"boys");
      Thread thread2=new Thread(aETC,"girls");
      thread2.setPriority(thread1.getPriority()+2);
      thread1.start();
      thread2.start();
      }
   }
```

The above program printed an output as shown below:

```
   :
   :
girls
girls
girls
girls
  :
  :
```

until the program was interrupted.

Thread Synchronization:

Threads that operate independently of their own are quite easy to program. It takes a little more skill to create a program in which there are a number of threads that must inter operate. Multiple threads in a method that accesses and modifies data can create erroneous results.

Synchronization in java is the capability of control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why we use Synchronization:

1.To prevent thread interference.

2.To prevent consistency problem.

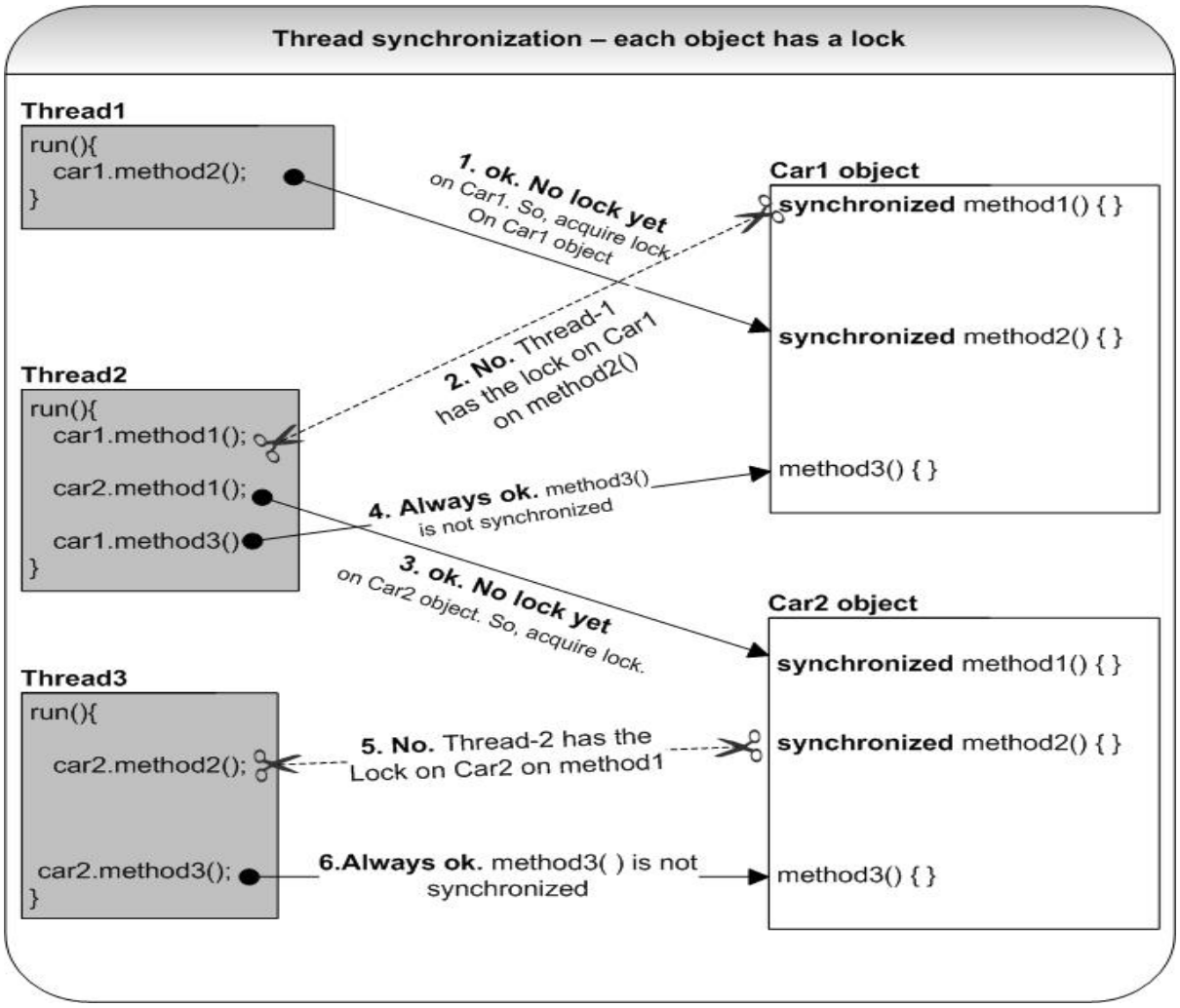
Note: Synchronized method is used to lock an object for any shared resource.

Synchronized Key Word

Any method that access and modifies data should be declared synchronized as follows :

```
Synchronized void modify data()
        {
         --------
         -------
        }
```

A synchronized method will not allow multiple threads in it.



Monitor:

The key *synchronized* puts a bottleneck around the method and allows only one thread to execute in it at a time. A method with a bottleneck is called a *monitor.*

You can label individual data items as synchronized like this:

```
synchronized(p)
    {
    x=p.getx();
    y=p.gety();
    }
```

This ensures that only one thread will be allowed inside the block that follows the synchronized keyword and also the object p is blocked, i.e. the other threads will not make changes in the object p. This provides finer granularity for mutual exclusion than an entire method.

Thread Synchronization

An Example

In the following program two threads are created. Each thread is depositing an amount of Rs 1000 into an account. The current balance in this account is Rs 1000. After depositing Rs 1000 by each thread the final balance should be Rs 3000. The following program shows the scenario in absence of proper thread handling

```
import java.io.*;
public class Deposit
{
      static int balance = 1000;
  public static void main(String[ ] args)
   {
            PrintWriter out = new PrintWriter(System.out,true);
           Account account = new Account(out);
           DepositThread first;
           DepositThread second;
           first = new DepositThread(account, 1000, "#1");
           second = new DepositThread(account, 1000, "\t\t\t\t#2");
           first.start( );
           second.start( );
           // wait for both the threads to finish
           try
              {
              first.join( );
              second.join( );
              } catch(InterruptedException e)
                    {    }
            out.println("Final balance is :" +balance);
   }
}
class Account
{
      PrintWriter out;
      public Account(PrintWriter out)
           {
              this.out = out;
              }
      public void deposit(int amount, String name)
           {
            int balance;
          out.println(name+" trying to deposit" +amount);
          out.println(name+" getting balance...");
          balance = getBalance( );
          out.println(name+" setting balance....");
          setBalance(balance);
          out.println(name+" new balance set to" + Deposit.balance);
          }
public int getBalance( )
      {
```

```
        try
                    {
                    Thread.sleep(5000);
                     } catch(InterruptedException e)
                        { }
                  return Deposit.balance;
        }


public void setBalance(int balance)
              {
                     try
                    {
                    Thread.sleep(5000);
                     } catch(InterruptedException e)
                         {}
                    Deposit.balance=balance;
               }
  }



class DepositThread extends Thread
{
       Account account;
       int amount;
       String message;
       public DepositThread(Account account, int amount, String message)
              {
               this.account = account;
               this.amount = amount;
               this.message = message;
               }


public void run( )
        {
          account.deposit(amount, message);
         }
   }
```

A typical output of the above program is shown below.

```
 #1 trying to deposit 1000
 #1 getting balanced.....
 #2 trying to deposit 10000
 #2 getting balance.....
 #1 balance got is 1000
 #1 setting balance.....
 #2 balance got is 1000
 #2 setting balance.....
```

#1 new balance set to 2000
#2 new balance set to 2000

Both the threads are depositing Rs 1000 each. Initial balance is Rs 1000. Therefore the anticipated final balance is Rs 3000. However our program is giving the final balance as Rs 2000. This means that an error has occurred. The reason for this error is the improper handling of threads.
The solution to this problem is to declare the deposit() method in the Account class synchronized as shown below.

```
public synchronized void deposit(int amount, String name)
```

A synchronized method will not allow more than one thread in that method at the same time.

With this modification, the above program will print the following output.

```
#1 trying to deposit 1000
#1 getting balance.....
#1 balance got is 1000
#1 setting balance.....
#1 new balance set to 2000
# 2 trying to deposit 1000
# 2 getting balance.....
# 2 balance got is 2000
# 2 setting balance.....
# 2 new balance set to 3000
Final balance is 3000
```

Another solution is to make a block of statement in the deposit( ) method synchronized on the called object as shown below:

```
public void deposit(int amount, String name)
    {
    int balance;
    out.primtln(name+" trying to deposit" +amount);
    synchronized(this)
          {
           out.println(name+"getting balance.....");
           balance = getBalance();
           out.println(name +" balance got is" +balance);
           balance+ =amount;
           out.println(name+" setting balance.....");
           setBalance(balance);
         }
  out.println(name+" new balance set to "+ Deposit.balance);
   }
```

When a block of statements in a method is declared as synchronized on an object, it will not allow more than one thread in the synchronized block. It will not allow any other thread in any of the methods of that object. If the synchronized block is in one of the methods of the object on which the block is synchronized, other threads are allowed in that method up to the synchronized block.
With the above modification our program prints the following output.

```
#1 trying to deposit 1000
```

#1 getting balance.....
#2 trying to deposit 1000
#1 balance got is 1000
#1 setting balance.....
#1 new balance set to 2000
#2 getting balance.....
#2 balance got is 2000
#2 setting balance.....
#2 new balance set to 3000
Final balance is 3000

Thread *first* starts and prints the following two lines.
#1 trying to deposit 1000
#1 getting balance
After printing these two lines thread *first* will goes to the getBalance( ) method where it will sleep for 5 seconds. Thread *second* now starts and calls the deposit( ) method of Account class with parameters, amount = 1000 and name = "#2". It prints the following line.
#2 trying to deposit 1000
Thread *second cannot* proceed further because another thread cannot enter the synchronized block. Thread *second* has to wait out side the synchronized block until thread *first* completes the synchronized block.

Semaphores– Semaphore (Counting Semaphore)
->acquire(), release()
->Mechanism to control access to a pool of shared resource, between multiple processes, threads
->Acts like a gate – for a limited access pool / lift – with a fixed capacity; some threads have to yield control, for the other threads to access the shared data;
->While Locks are exclusive, semaphores are not
-->Eg: Fixed no. of meeting rooms – with controlled access

Mutexes – Same as a binary semaphore (lock - yes/no), but across processes
Normally mutexes has owners
Typical usage – to ensure single instance of an application / process