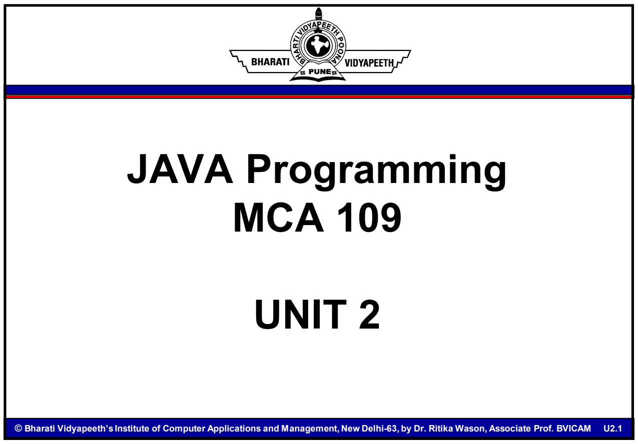
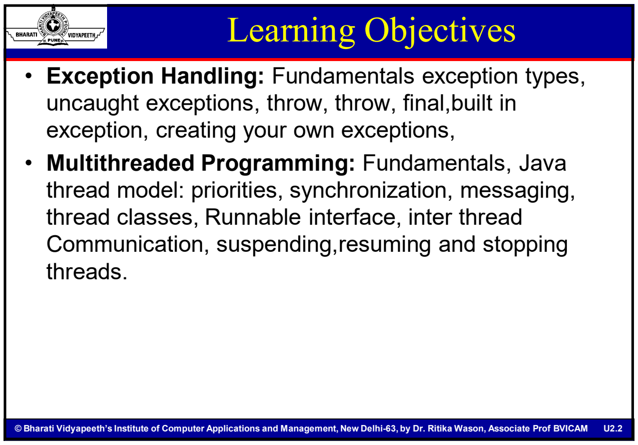


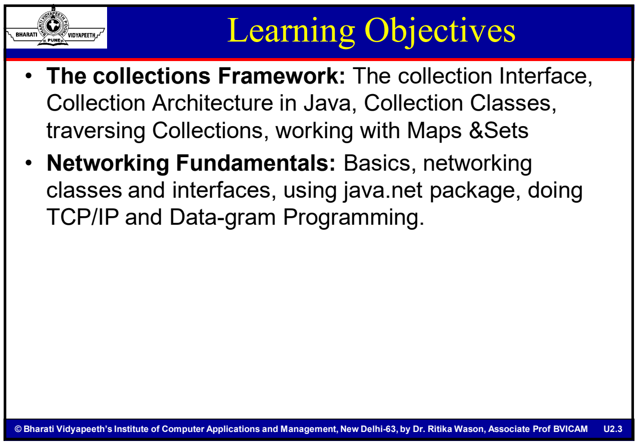
JAVA Programming
MCA 109

UNIT 2



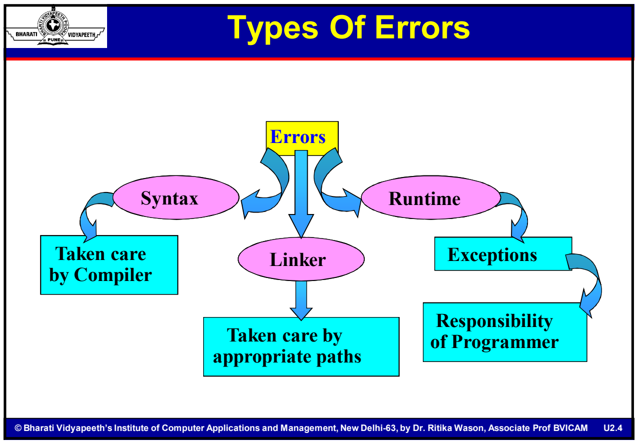
Learning Objectives

- **Exception Handling:** Fundamentals exception types, uncaught exceptions, throw, throw, final,built in exception, creating your own exceptions,
- **Multithreaded Programming:** Fundamentals, Java thread model: priorities, synchronization, messaging, thread classes, Runnable interface, inter thread Communication, suspending,resuming and stopping threads.



Learning Objectives

- **The collections Framework:** The collection Interface, Collection Architecture in Java, Collection Classes, traversing Collections, working with Maps &Sets
- **Networking Fundamentals:** Basics, networking classes and interfaces, using java.net package, doing TCP/IP and Data-gram Programming.



Types Of Errors

Errors

Syntax

Taken care by Compiler

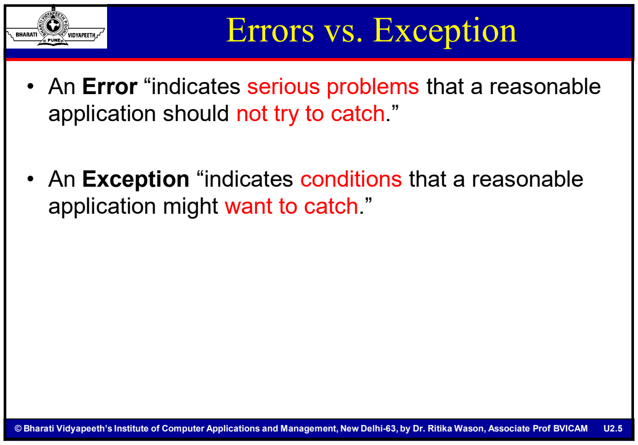
Linker

Taken care by appropriate paths

Runtime

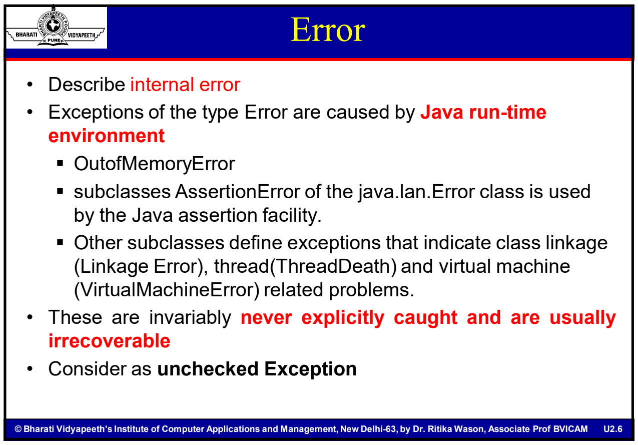
Exceptions

Responsibility of Programmer



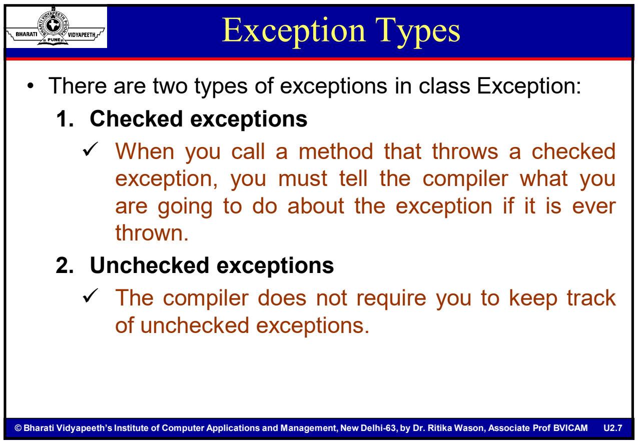
Errors vs. Exception

- An **Error** "indicates serious problems that a reasonable application should not try to catch."
- An **Exception** "indicates conditions that a reasonable application might want to catch."



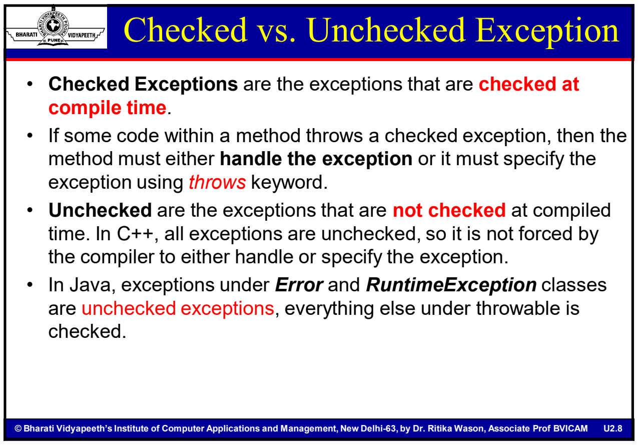
Error

- Describe internal error
- Exceptions of the type Error are caused by **Java run-time environment**
  - OutofMemoryError
  - subclasses AssertionError of the java.lan.Error class is used by the Java assertion facility.
  - Other subclasses define exceptions that indicate class linkage (Linkage Error), thread(ThreadDeath) and virtual machine (VirtualMachineError) related problems.
- These are invariably **never explicitly caught and are usually irrecoverable**
- Consider as **unchecked Exception**



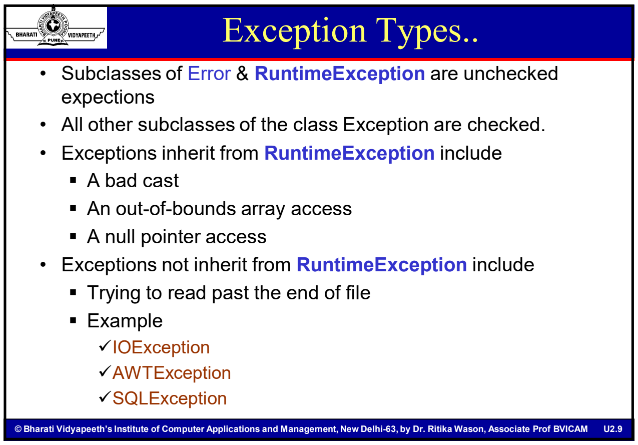
Exception Types

- There are two types of exceptions in class Exception:
  1. **Checked exceptions**
     - ✓ When you call a method that throws a checked exception, you must tell the compiler what you are going to do about the exception if it is ever thrown.
  2. **Unchecked exceptions**
     - ✓ The compiler does not require you to keep track of unchecked exceptions.



Checked vs. Unchecked Exception

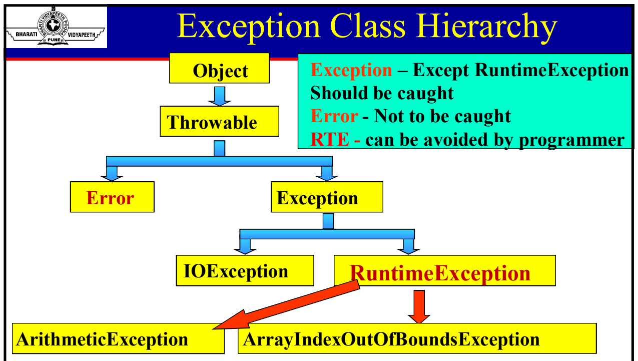
- **Checked Exceptions** are the exceptions that are **checked at compile time**.
- If some code within a method throws a checked exception, then the method must either **handle the exception** or it must specify the exception using *throws* keyword.
- **Unchecked** are the exceptions that are **not checked** at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception.
- In Java, exceptions under ***Error*** and ***RuntimeException*** classes are unchecked exceptions, everything else under throwable is checked.



Exception Types..

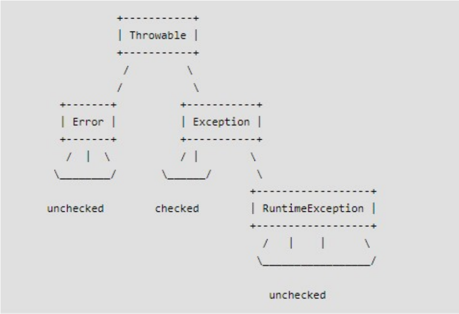
- Subclasses of Error & **RuntimeException** are unchecked expections
- All other subclasses of the class Exception are checked.
- Exceptions inherit from **RuntimeException** include
  - A bad cast
  - An out-of-bounds array access
  - A null pointer access
- Exceptions not inherit from **RuntimeException** include
  - Trying to read past the end of file
  - Example
    - ✓IOException
    - ✓AWTException
    - ✓SQLException

Exception Class Hierarchy



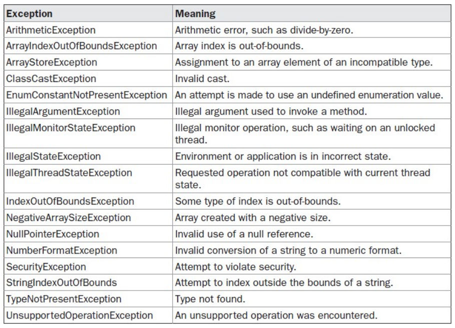
Exception – Except RuntimeException Should be caught
Error - Not to be caught
RTE - can be avoided by programmer

Exception Hierarchy

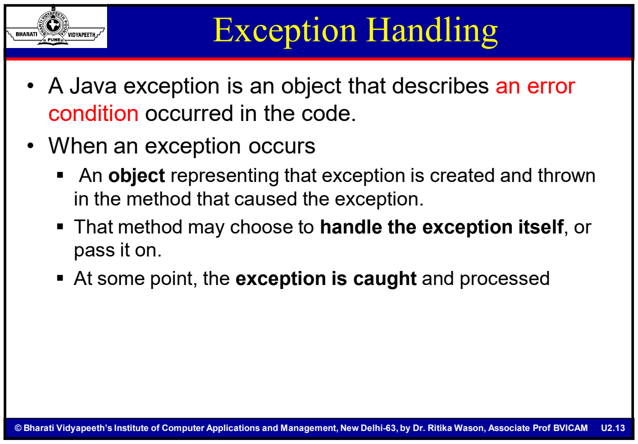


```
                  +-----------+
                  | Throwable |
                  +-----------+
                   /         \
                  /           \
         +-------+          +-----------+
         | Error |          | Exception |
         +-------+          +-----------+
          /  |  \            / |        \
        \_______/         \_____/        \
                                       +------------------+
      unchecked         checked        | RuntimeException |
                                       +------------------+
                                         /   |    |      \
                                        \__________________/

                                          unchecked
```

Built-in Exceptions

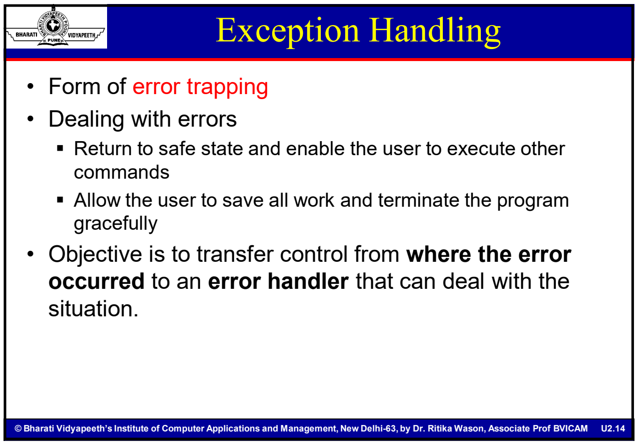


Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.



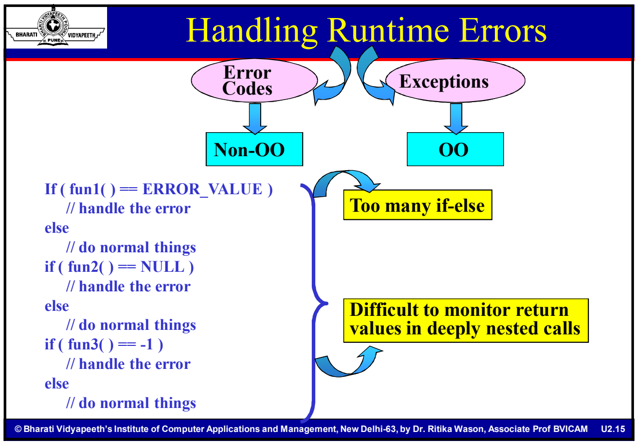
Exception Handling

- A Java exception is an object that describes an error condition occurred in the code.
- When an exception occurs
  - An **object** representing that exception is created and thrown in the method that caused the exception.
  - That method may choose to **handle the exception itself**, or pass it on.
  - At some point, the **exception is caught** and processed



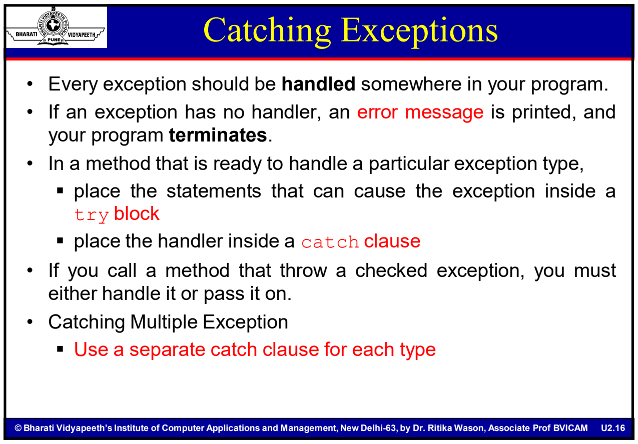
Exception Handling

- Form of error trapping
- Dealing with errors
  - Return to safe state and enable the user to execute other commands
  - Allow the user to save all work and terminate the program gracefully
- Objective is to transfer control from **where the error occurred** to an **error handler** that can deal with the situation.



Handling Runtime Errors

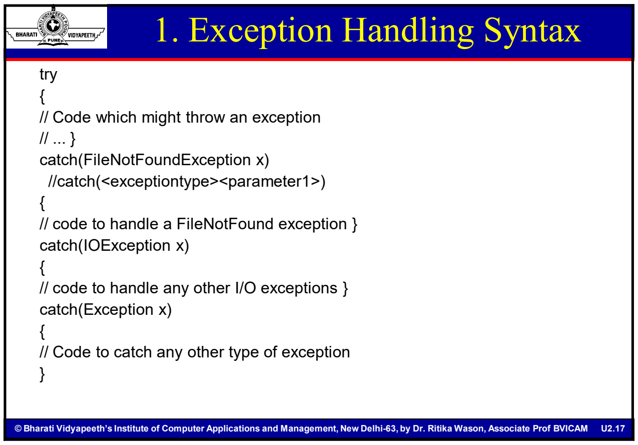
```
If ( fun1( ) == ERROR_VALUE )
    // handle the error
else
    // do normal things
if ( fun2( ) == NULL )
    // handle the error
else
    // do normal things
if ( fun3( ) == -1 )
    // handle the error
else
    // do normal things
```



Catching Exceptions

- Every exception should be **handled** somewhere in your program.
- If an exception has no handler, an error message is printed, and your program **terminates**.
- In a method that is ready to handle a particular exception type,
  - place the statements that can cause the exception inside a `try` block
  - place the handler inside a `catch` clause
- If you call a method that throw a checked exception, you must either handle it or pass it on.
- Catching Multiple Exception
  - Use a separate catch clause for each type

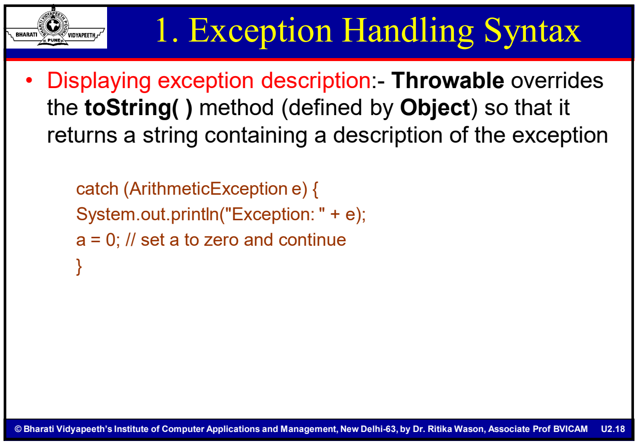
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.16



1. Exception Handling Syntax

```
try
{
// Code which might throw an exception
// ... }
catch(FileNotFoundException x)
  //catch(<exceptiontype><parameter1>)
{
// code to handle a FileNotFound exception }
catch(IOException x)
{
// code to handle any other I/O exceptions }
catch(Exception x)
{
// Code to catch any other type of exception
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.17

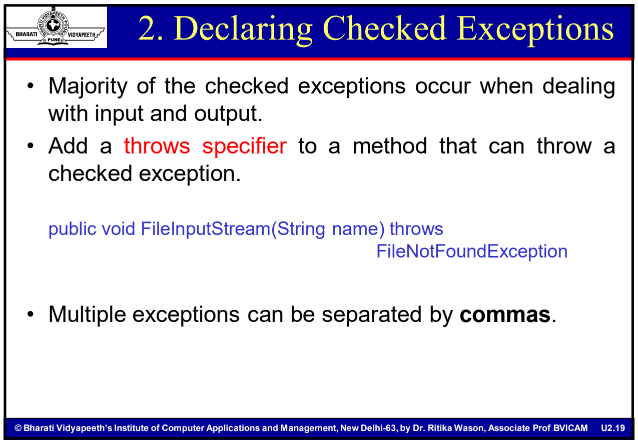


1. Exception Handling Syntax

- Displaying exception description:- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.18



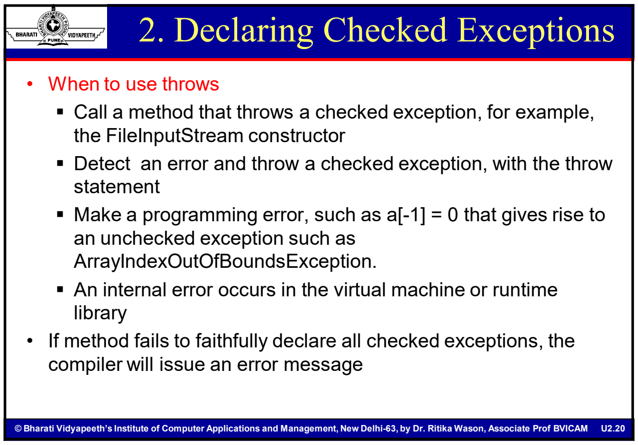
2. Declaring Checked Exceptions

- Majority of the checked exceptions occur when dealing with input and output.
- Add a throws specifier to a method that can throw a checked exception.

```
public void FileInputStream(String name) throws
                                   FileNotFoundException
```

- Multiple exceptions can be separated by **commas**.

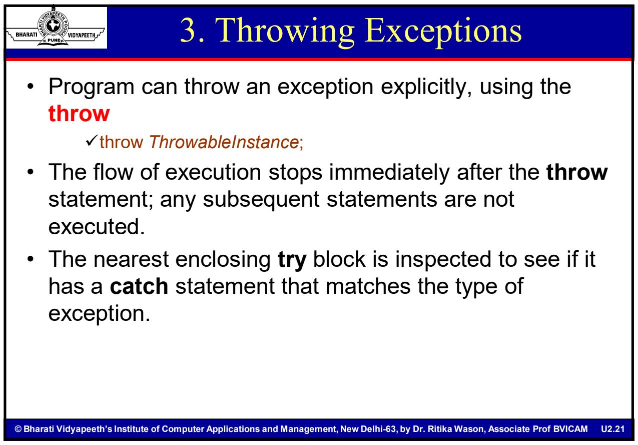
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.19



2. Declaring Checked Exceptions

- When to use throws
  - Call a method that throws a checked exception, for example, the FileInputStream constructor
  - Detect an error and throw a checked exception, with the throw statement
  - Make a programming error, such as a[-1] = 0 that gives rise to an unchecked exception such as ArrayIndexOutOfBoundsException.
  - An internal error occurs in the virtual machine or runtime library
- If method fails to faithfully declare all checked exceptions, the compiler will issue an error message

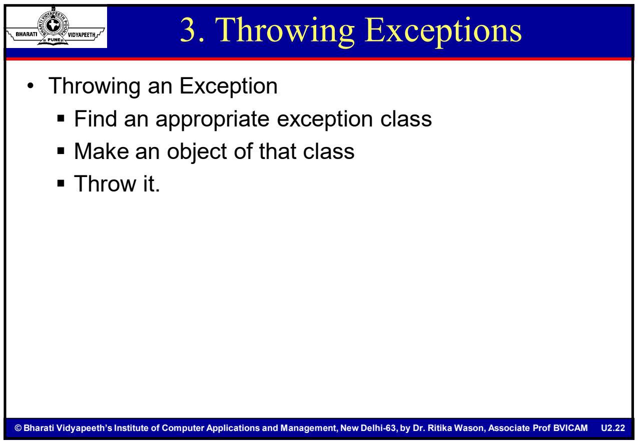
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.20



3. Throwing Exceptions

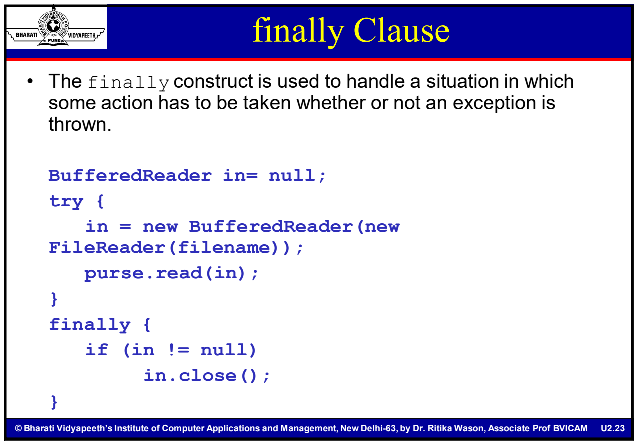
- Program can throw an exception explicitly, using the **throw**
  - ✓throw *ThrowableInstance*;
- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.21



3. Throwing Exceptions

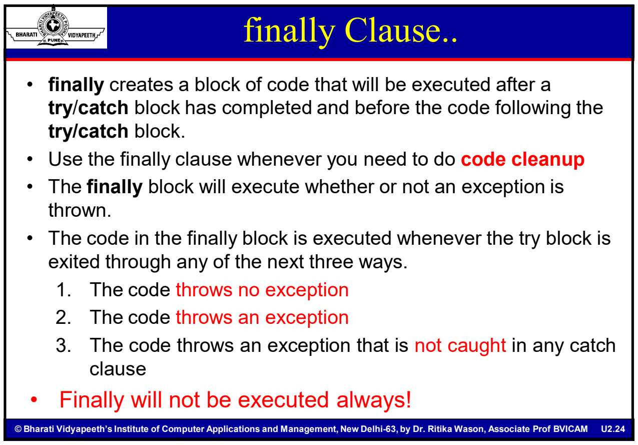
- Throwing an Exception
  - Find an appropriate exception class
  - Make an object of that class
  - Throw it.



finally Clause

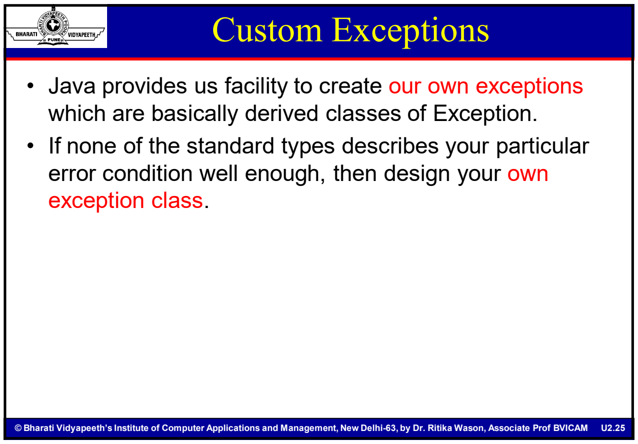
- The `finally` construct is used to handle a situation in which some action has to be taken whether or not an exception is thrown.

```
BufferedReader in= null;
try {
   in = new BufferedReader(new
FileReader(filename));
   purse.read(in);
}
finally {
   if (in != null)
        in.close();
}
```



finally Clause..

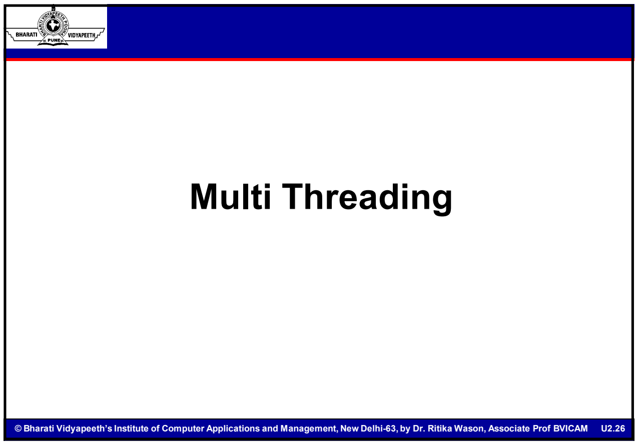
- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- Use the finally clause whenever you need to do **code cleanup**
- The **finally** block will execute whether or not an exception is thrown.
- The code in the finally block is executed whenever the try block is exited through any of the next three ways.
  1. The code throws no exception
  2. The code throws an exception
  3. The code throws an exception that is not caught in any catch clause
- Finally will not be executed always!



Custom Exceptions

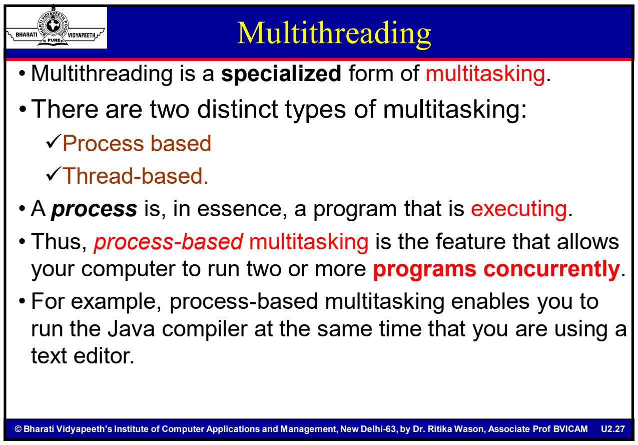
- Java provides us facility to create our own exceptions which are basically derived classes of Exception.
- If none of the standard types describes your particular error condition well enough, then design your own exception class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.25



Multi Threading

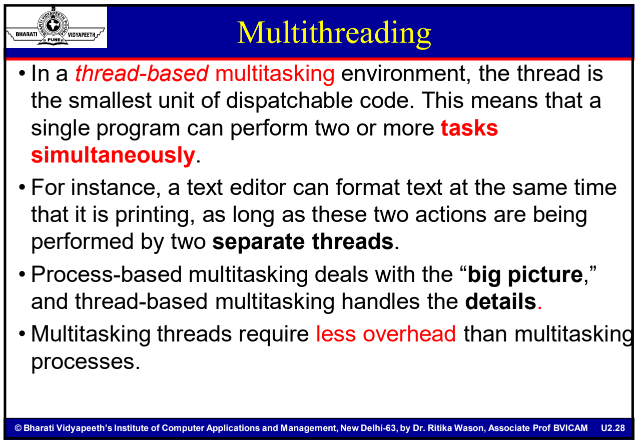
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.26



Multithreading

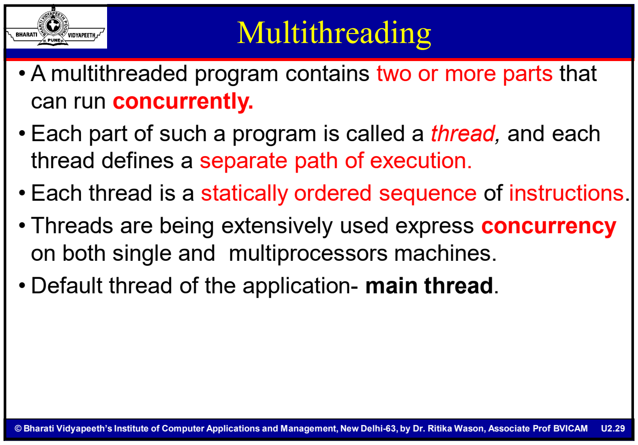
- Multithreading is a **specialized** form of multitasking.
- There are two distinct types of multitasking:
  - ✓Process based
  - ✓Thread-based.
- A ***process*** is, in essence, a program that is executing.
- Thus, *process-based* multitasking is the feature that allows your computer to run two or more **programs concurrently**.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.27



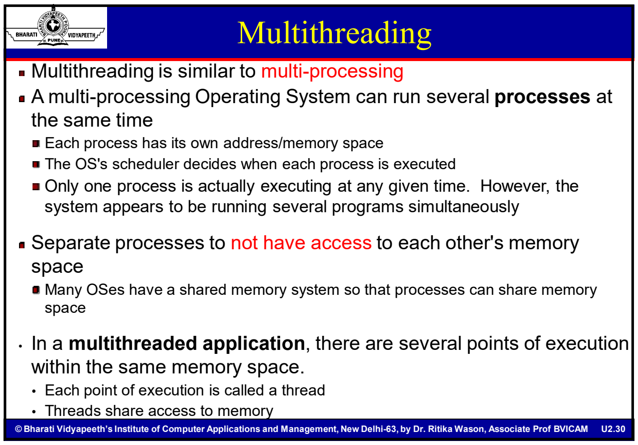
Multithreading

- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more **tasks simultaneously**.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two **separate threads**.
- Process-based multitasking deals with the “**big picture**,” and thread-based multitasking handles the **details**.
- Multitasking threads require less overhead than multitasking processes.



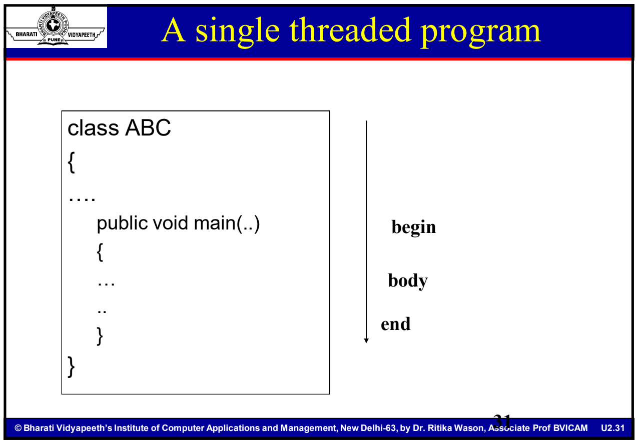
Multithreading

- 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.
- Each thread is a statically ordered sequence of instructions.
- Threads are being extensively used express **concurrency** on both single and multiprocessors machines.
- Default thread of the application- **main thread**.



Multithreading

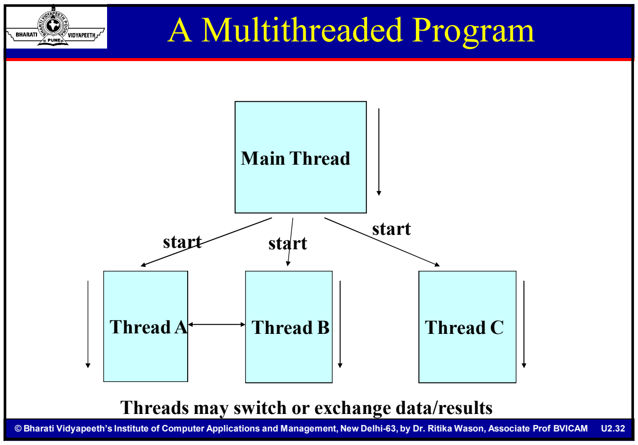
- Multithreading is similar to multi-processing
- A multi-processing Operating System can run several **processes** at the same time
  - Each process has its own address/memory space
  - The OS's scheduler decides when each process is executed
  - Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- Separate processes to not have access to each other's memory space
  - Many OSes have a shared memory system so that processes can share memory space
- In a **multithreaded application**, there are several points of execution within the same memory space.
  - Each point of execution is called a thread
  - Threads share access to memory



A single threaded program

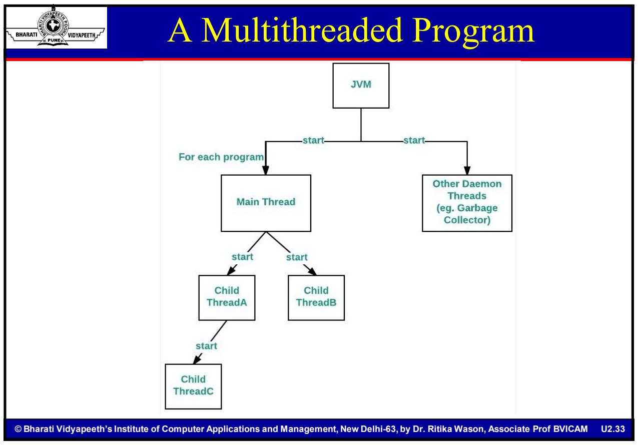
```
class ABC
{
….
     public void main(..)
     {
     …
     ..
     }
}
```

begin
body
end

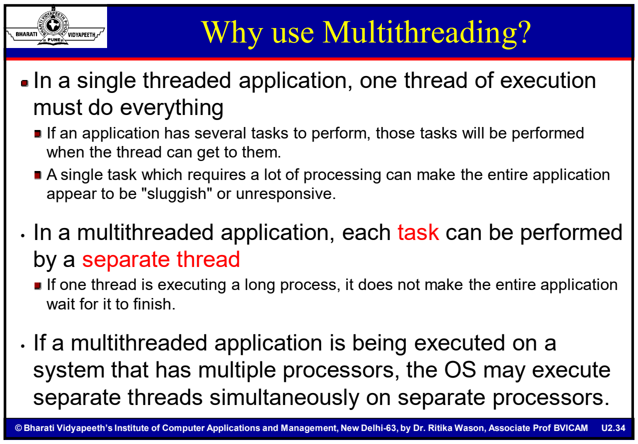


A Multithreaded Program

Threads may switch or exchange data/results

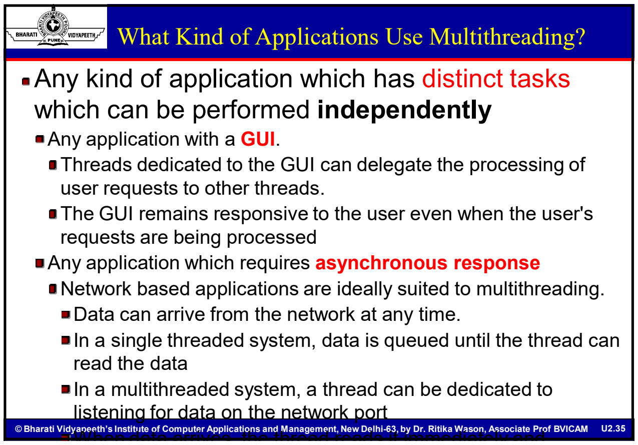


A Multithreaded Program



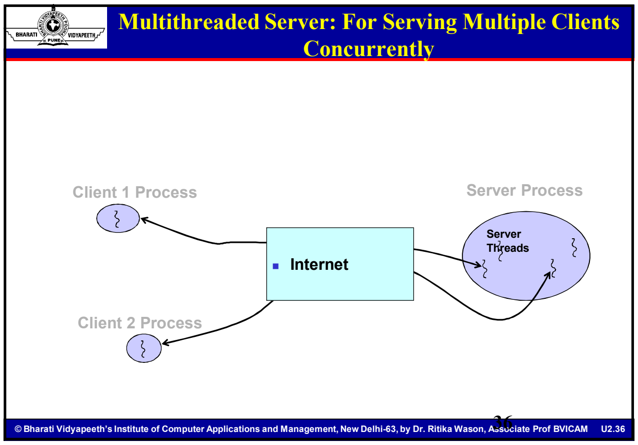
Why use Multithreading?

- In a single threaded application, one thread of execution must do everything
  - If an application has several tasks to perform, those tasks will be performed when the thread can get to them.
  - A single task which requires a lot of processing can make the entire application appear to be "sluggish" or unresponsive.
- In a multithreaded application, each task can be performed by a separate thread
  - If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

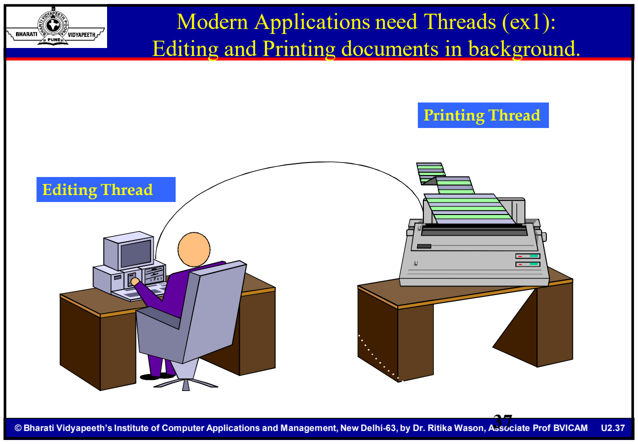


What Kind of Applications Use Multithreading?

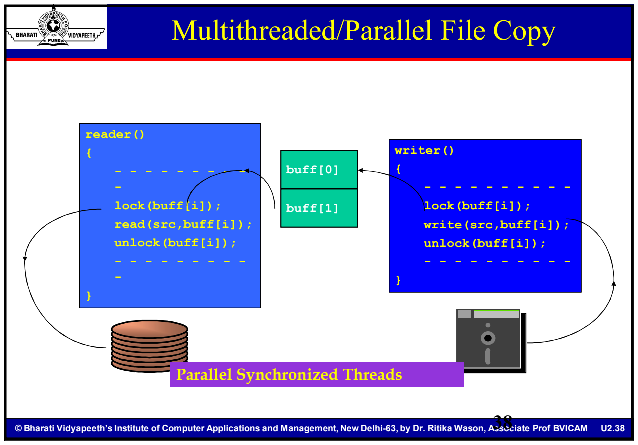
- Any kind of application which has distinct tasks which can be performed **independently**
  - Any application with a **GUI**.
    - Threads dedicated to the GUI can delegate the processing of user requests to other threads.
    - The GUI remains responsive to the user even when the user's requests are being processed
  - Any application which requires **asynchronous response**
    - Network based applications are ideally suited to multithreading.
      - Data can arrive from the network at any time.
      - In a single threaded system, data is queued until the thread can read the data
      - In a multithreaded system, a thread can be dedicated to listening for data on the network port



Multithreaded Server: For Serving Multiple Clients Concurrently



Modern Applications need Threads (ex1): Editing and Printing documents in background.

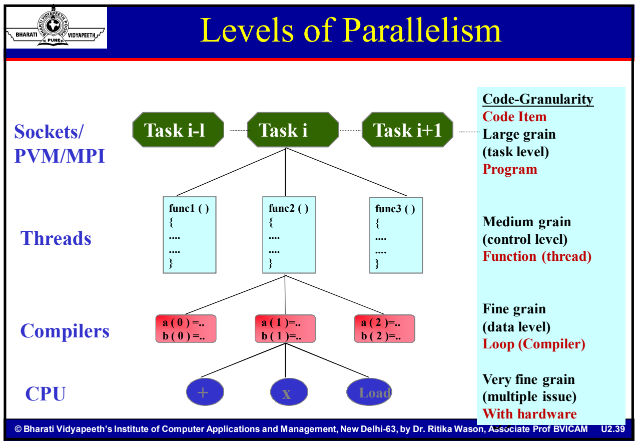


Multithreaded/Parallel File Copy

```
reader()
{
    - - - - - - - - - -
    lock(buff[i]);
    read(src,buff[i]);
    unlock(buff[i]);
    - - - - - - - - - -
}
```

```
writer()
{
    - - - - - - - - - -
    lock(buff[i]);
    write(src,buff[i]);
    unlock(buff[i]);
    - - - - - - - - - -
}
```

Parallel Synchronized Threads



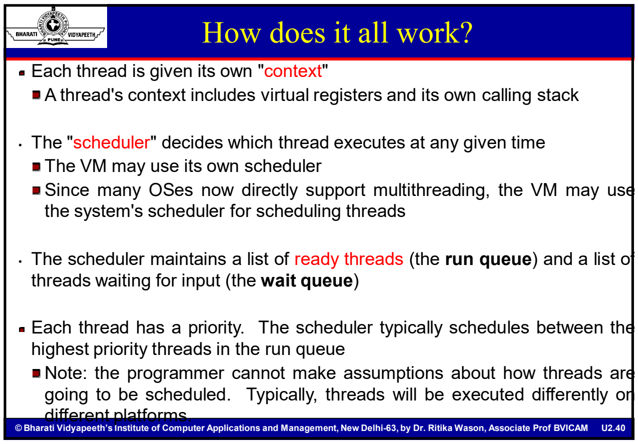
Levels of Parallelism

Code-Granularity
Code Item
Large grain
(task level)
Program

Medium grain
(control level)
Function (thread)

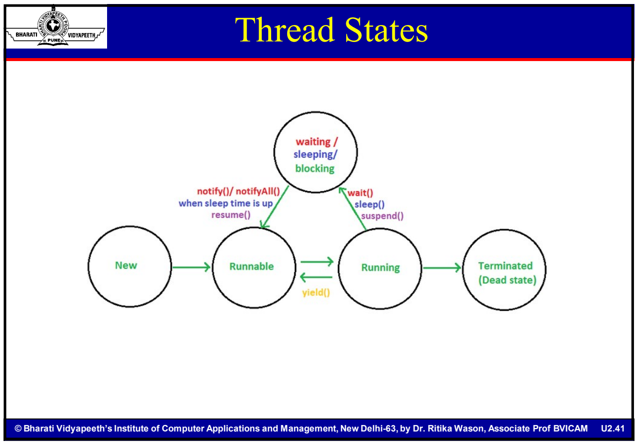
Fine grain
(data level)
Loop (Compiler)

Very fine grain
(multiple issue)
With hardware

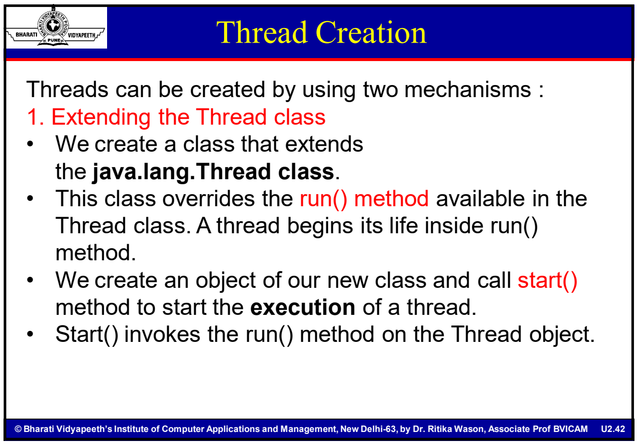


How does it all work?

- Each thread is given its own "context"
  - A thread's context includes virtual registers and its own calling stack
- The "scheduler" decides which thread executes at any given time
  - The VM may use its own scheduler
  - Since many OSes now directly support multithreading, the VM may use the system's scheduler for scheduling threads
- The scheduler maintains a list of ready threads (the **run queue**) and a list of threads waiting for input (the **wait queue**)
- Each thread has a priority. The scheduler typically schedules between the highest priority threads in the run queue
  - Note: the programmer cannot make assumptions about how threads are going to be scheduled. Typically, threads will be executed differently on different platforms.



Thread States

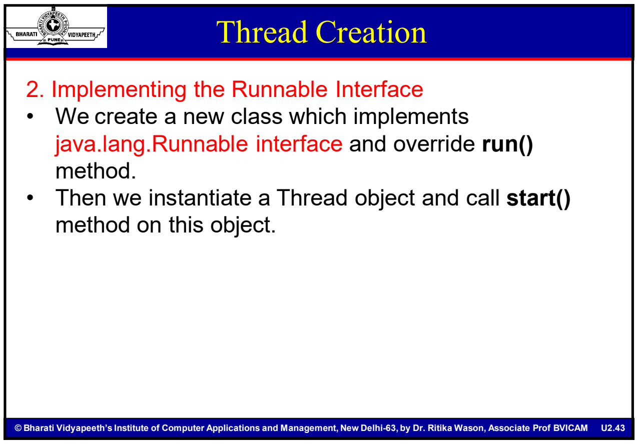


Thread Creation

Threads can be created by using two mechanisms :

1. Extending the Thread class

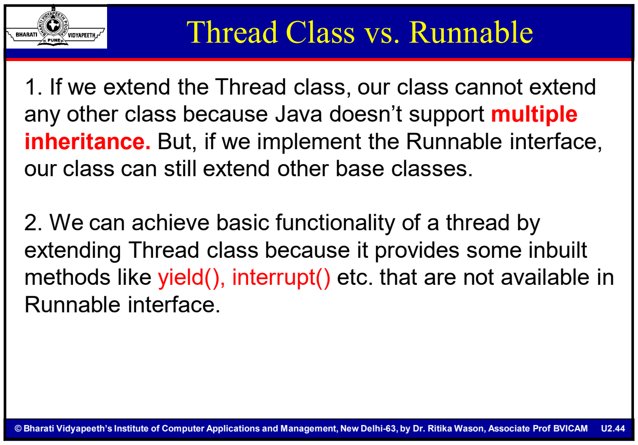
- We create a class that extends the **java.lang.Thread class**.
- This class overrides the run() method available in the Thread class. A thread begins its life inside run() method.
- We create an object of our new class and call start() method to start the **execution** of a thread.
- Start() invokes the run() method on the Thread object.



Thread Creation

2. Implementing the Runnable Interface

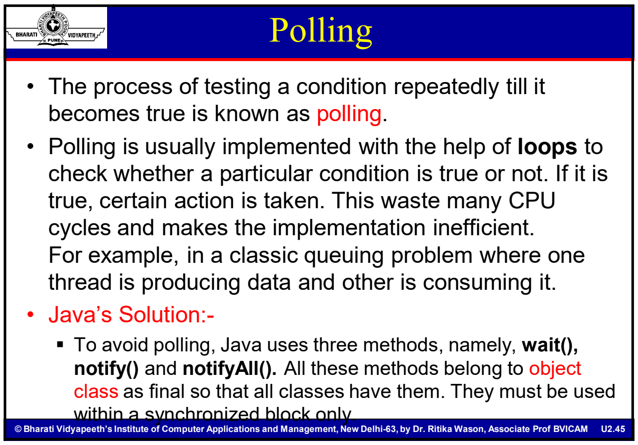
- We create a new class which implements java.lang.Runnable interface and override **run()** method.
- Then we instantiate a Thread object and call **start()** method on this object.



Thread Class vs. Runnable

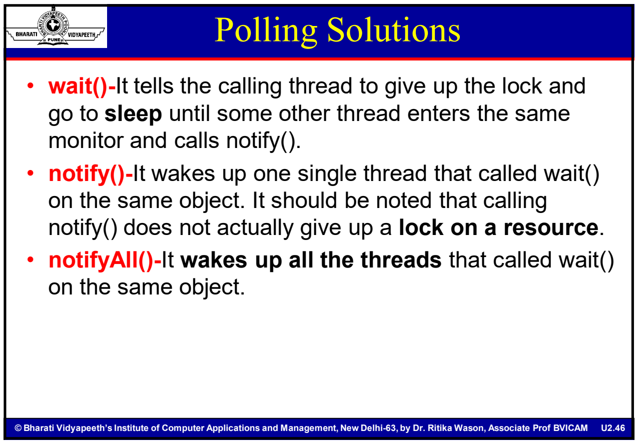
1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support **multiple inheritance.** But, if we implement the Runnable interface, our class can still extend other base classes.

2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.



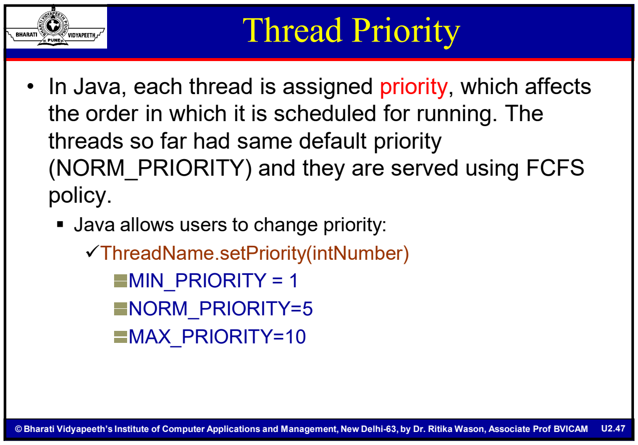
Polling

- The process of testing a condition repeatedly till it becomes true is known as polling.
- Polling is usually implemented with the help of **loops** to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient. For example, in a classic queuing problem where one thread is producing data and other is consuming it.
- Java's Solution:-
  - To avoid polling, Java uses three methods, namely, **wait(), notify()** and **notifyAll().** All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.



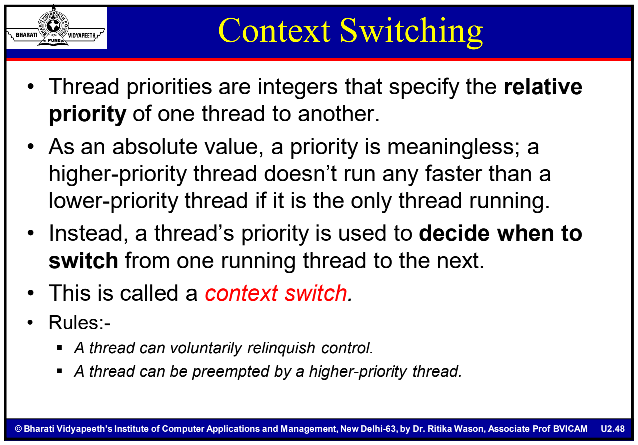
Polling Solutions

- **wait()-**It tells the calling thread to give up the lock and go to **sleep** until some other thread enters the same monitor and calls notify().
- **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a **lock on a resource**.
- **notifyAll()-**It **wakes up all the threads** that called wait() on the same object.



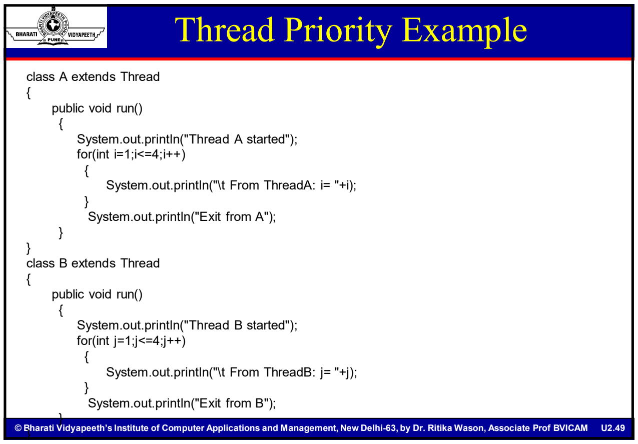
Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ✓ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10



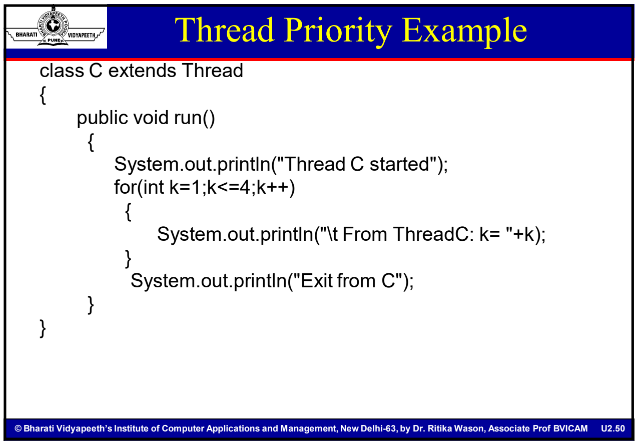
Context Switching

- Thread priorities are integers that specify the **relative priority** of one thread to another.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, a thread's priority is used to **decide when to switch** from one running thread to the next.
- This is called a *context switch*.
- Rules:-
  - *A thread can voluntarily relinquish control.*
  - *A thread can be preempted by a higher-priority thread.*



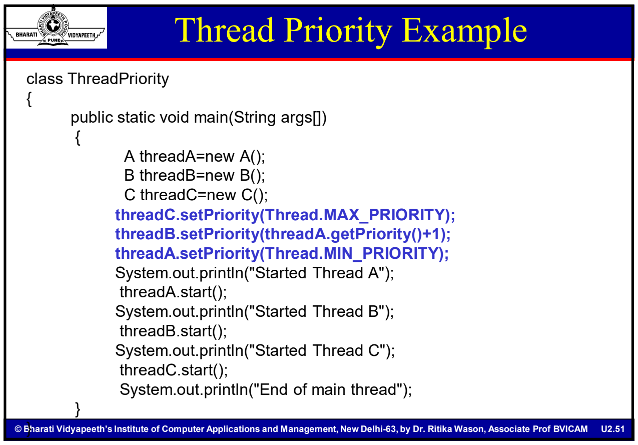
Thread Priority Example

```
class A extends Thread
{
     public void run()
      {
          System.out.println("Thread A started");
          for(int i=1;i<=4;i++)
           {
                System.out.println("\t From ThreadA: i= "+i);
           }
            System.out.println("Exit from A");
      }
}
class B extends Thread
{
     public void run()
      {
          System.out.println("Thread B started");
          for(int j=1;j<=4;j++)
           {
                System.out.println("\t From ThreadB: j= "+j);
           }
            System.out.println("Exit from B");
      }
```



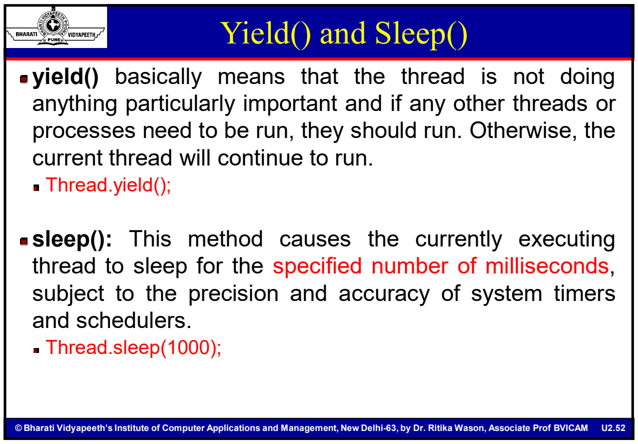
Thread Priority Example

```
class C extends Thread
{
     public void run()
      {
          System.out.println("Thread C started");
          for(int k=1;k<=4;k++)
           {
                System.out.println("\t From ThreadC: k= "+k);
           }
            System.out.println("Exit from C");
      }
}
```



Thread Priority Example

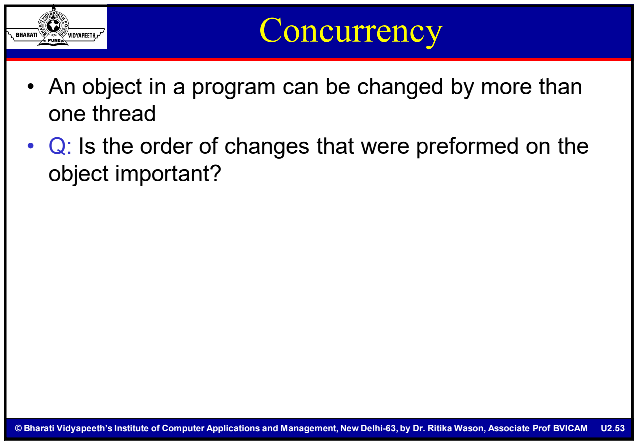
```
class ThreadPriority
{
        public static void main(String args[])
        {
                A threadA=new A();
                B threadB=new B();
                C threadC=new C();
              threadC.setPriority(Thread.MAX_PRIORITY);
              threadB.setPriority(threadA.getPriority()+1);
              threadA.setPriority(Thread.MIN_PRIORITY);
              System.out.println("Started Thread A");
               threadA.start();
              System.out.println("Started Thread B");
               threadB.start();
              System.out.println("Started Thread C");
               threadC.start();
               System.out.println("End of main thread");
        }
```



Yield() and Sleep()

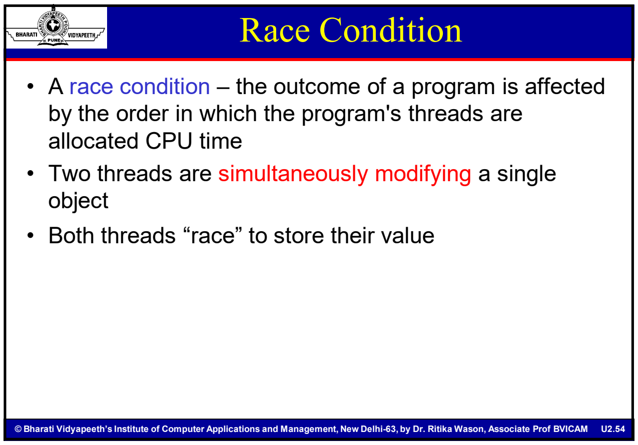
- **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.
  - Thread.yield();

- **sleep():** This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
  - Thread.sleep(1000);



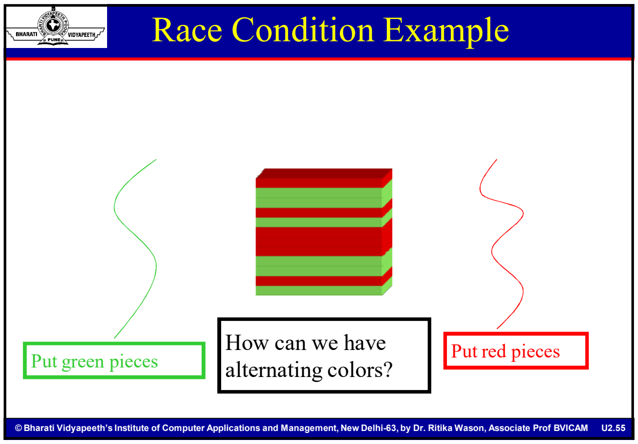
Concurrency

- An object in a program can be changed by more than one thread
- Q: Is the order of changes that were preformed on the object important?



Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object
- Both threads "race" to store their value

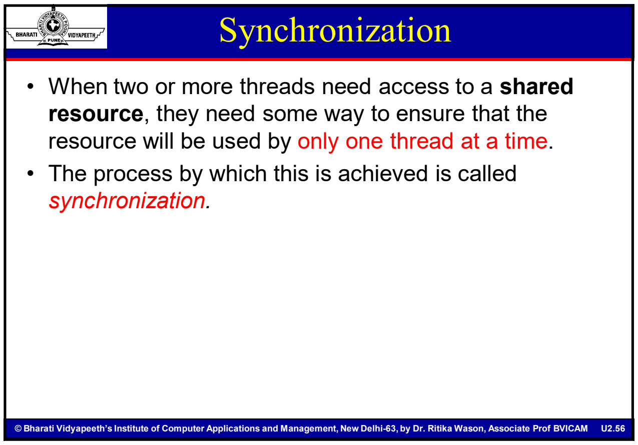


Race Condition Example

Put green pieces

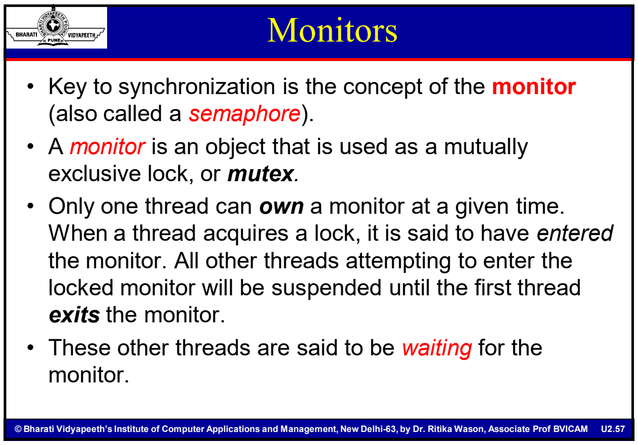
How can we have alternating colors?

Put red pieces



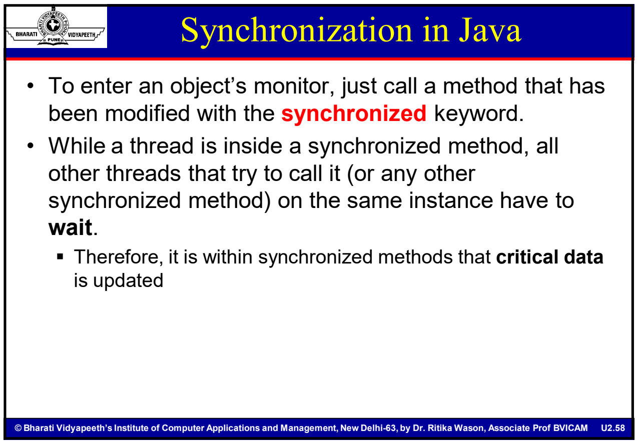
Synchronization

- When two or more threads need access to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called *synchronization*.



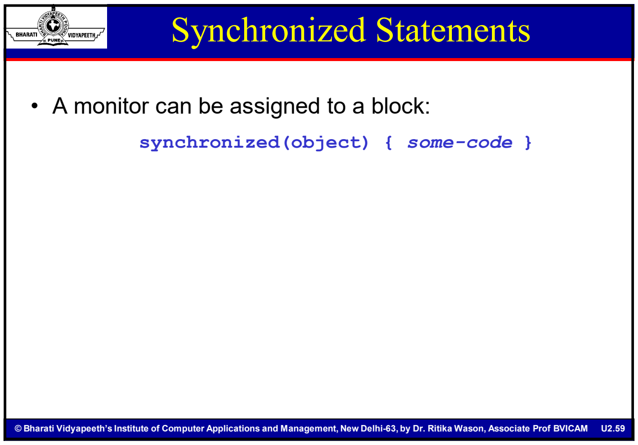
Monitors

- Key to synchronization is the concept of the **monitor** (also called a *semaphore*).
- A *monitor* is an object that is used as a mutually exclusive lock, or ***mutex***.
- Only one thread can ***own*** a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor.
- These other threads are said to be *waiting* for the monitor.



Synchronization in Java

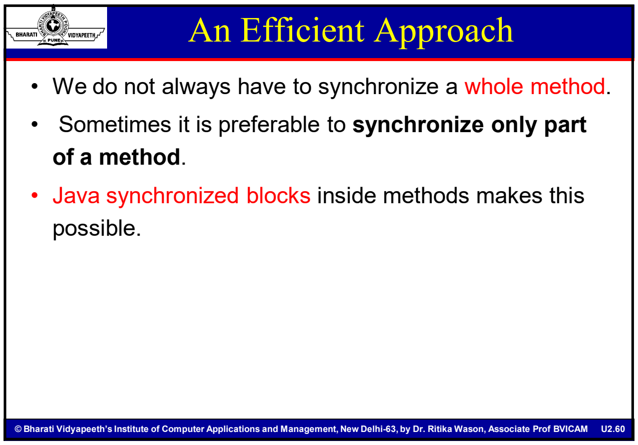
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to **wait**.
  - Therefore, it is within synchronized methods that **critical data** is updated



Synchronized Statements

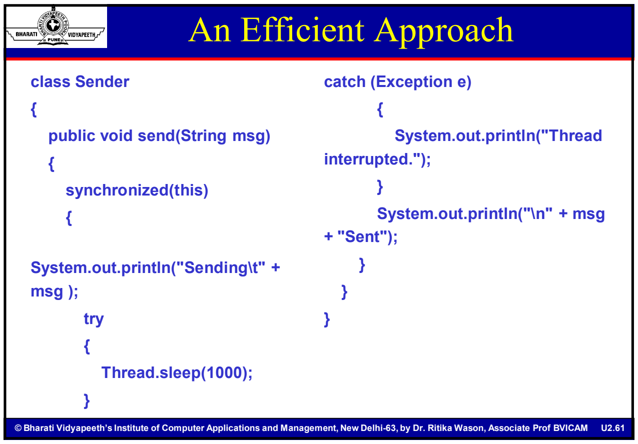
- A monitor can be assigned to a block:

```
synchronized(object) { some-code }
```



An Efficient Approach

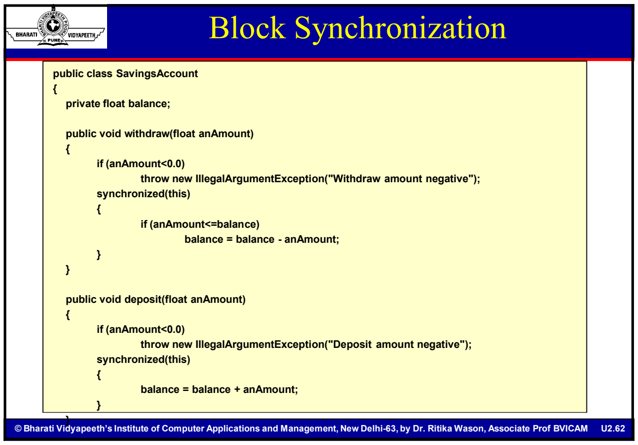
- We do not always have to synchronize a whole method.
- Sometimes it is preferable to **synchronize only part of a method**.
- Java synchronized blocks inside methods makes this possible.



An Efficient Approach

```
class Sender
{
   public void send(String msg)
   {
      synchronized(this)
      {
System.out.println("Sending\t" + msg );
         try
         {
            Thread.sleep(1000);
         }
catch (Exception e)
         {
            System.out.println("Thread interrupted.");
         }
         System.out.println("\n" + msg + "Sent");
      }
   }
}
```

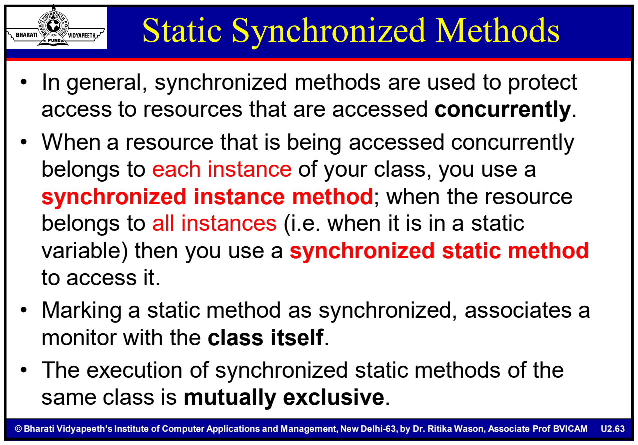
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.61



Block Synchronization

```
public class SavingsAccount
{
   private float balance;

   public void withdraw(float anAmount)
   {
         if (anAmount<0.0)
               throw new IllegalArgumentException("Withdraw amount negative");
         synchronized(this)
         {
               if (anAmount<=balance)
                     balance = balance - anAmount;
         }
   }

   public void deposit(float anAmount)
   {
         if (anAmount<0.0)
               throw new IllegalArgumentException("Deposit amount negative");
         synchronized(this)
         {
               balance = balance + anAmount;
         }
   }
```

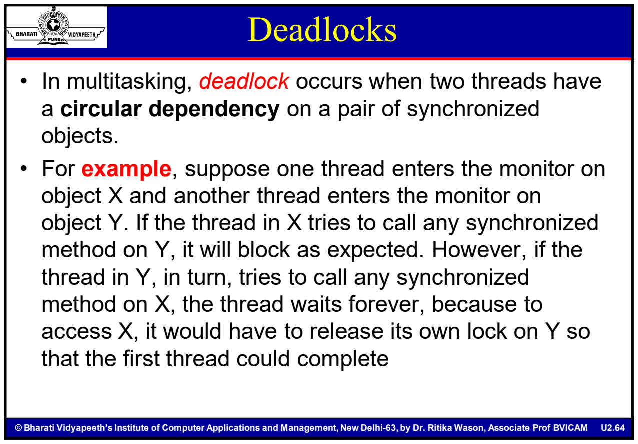
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.62



Static Synchronized Methods

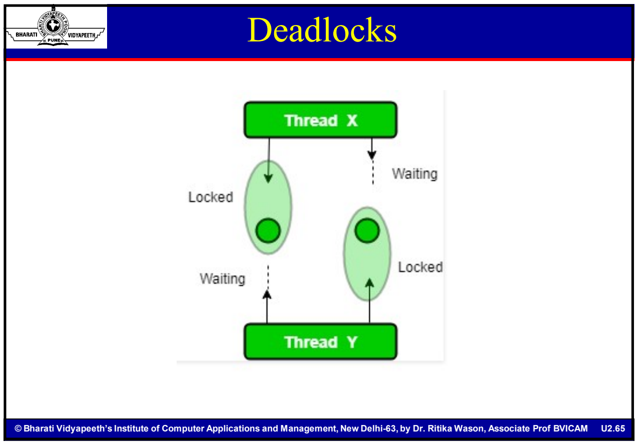
- In general, synchronized methods are used to protect access to resources that are accessed **concurrently**.
- When a resource that is being accessed concurrently belongs to each instance of your class, you use a **synchronized instance method**; when the resource belongs to all instances (i.e. when it is in a static variable) then you use a **synchronized static method** to access it.
- Marking a static method as synchronized, associates a monitor with the **class itself**.
- The execution of synchronized static methods of the same class is **mutually exclusive**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.63

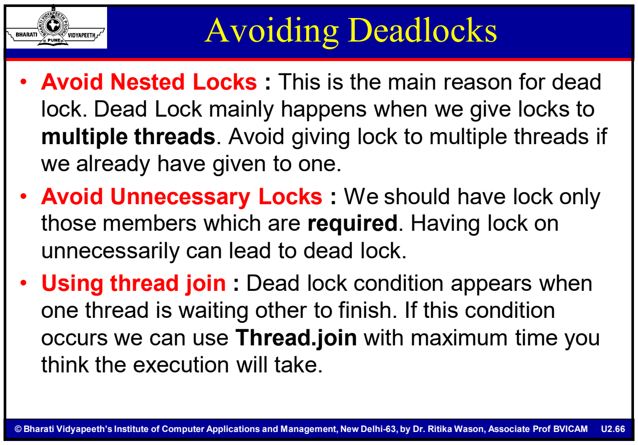


Deadlocks

- In multitasking, *deadlock* occurs when two threads have a **circular dependency** on a pair of synchronized objects.
- For **example**, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete

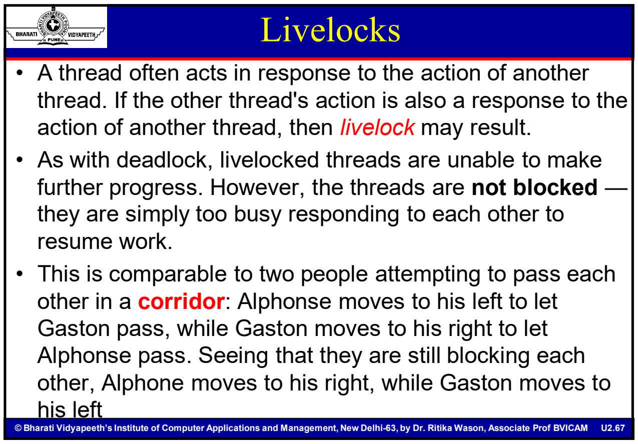


Deadlocks



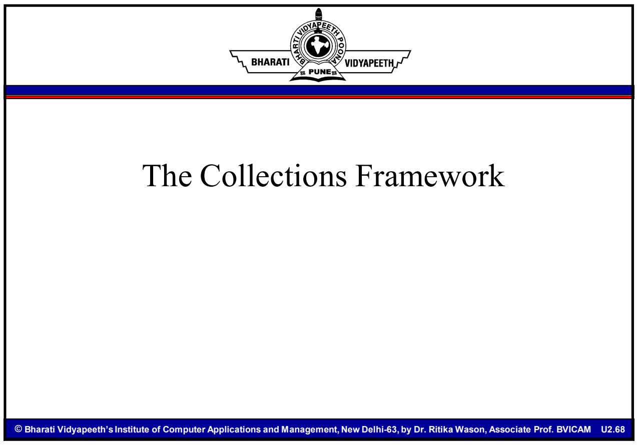
Avoiding Deadlocks

- **Avoid Nested Locks :** This is the main reason for dead lock. Dead Lock mainly happens when we give locks to **multiple threads**. Avoid giving lock to multiple threads if we already have given to one.
- **Avoid Unnecessary Locks :** We should have lock only those members which are **required**. Having lock on unnecessarily can lead to dead lock.
- **Using thread join :** Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use **Thread.join** with maximum time you think the execution will take.

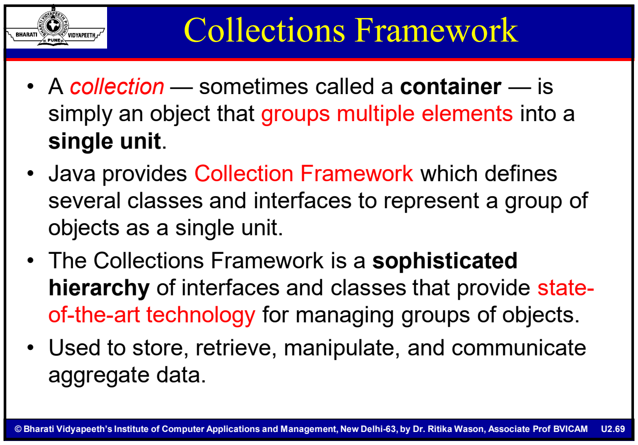


Livelocks

- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result.
- As with deadlock, livelocked threads are unable to make further progress. However, the threads are **not blocked** — they are simply too busy responding to each other to resume work.
- This is comparable to two people attempting to pass each other in a **corridor**: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left

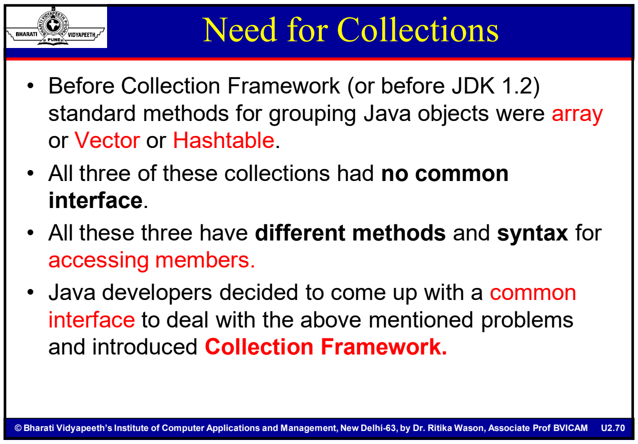


The Collections Framework



Collections Framework

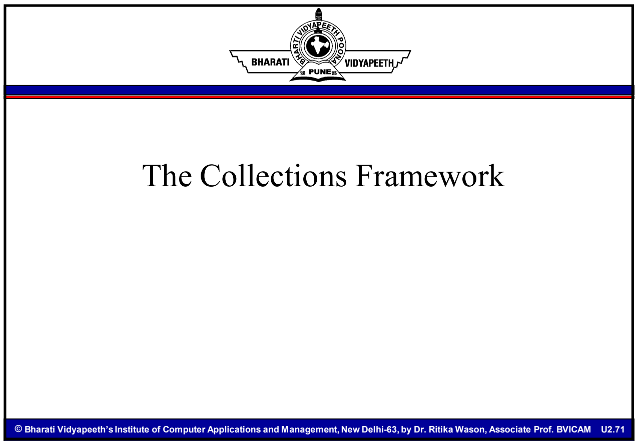
- A *collection* — sometimes called a **container** — is simply an object that groups multiple elements into a **single unit**.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collections Framework is a **sophisticated hierarchy** of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Used to store, retrieve, manipulate, and communicate aggregate data.



Need for Collections

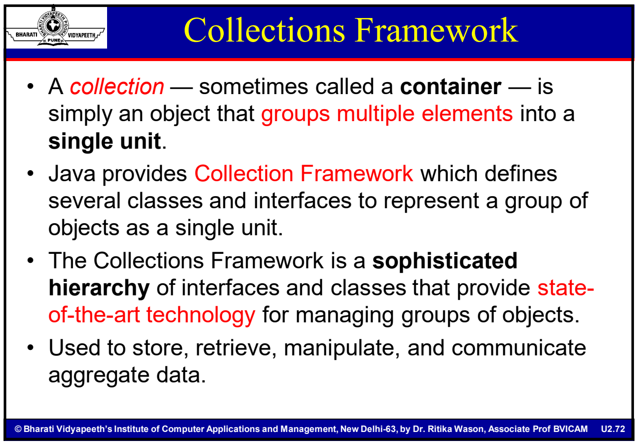
- Before Collection Framework (or before JDK 1.2) standard methods for grouping Java objects were array or Vector or Hashtable.
- All three of these collections had **no common interface**.
- All these three have **different methods** and **syntax** for accessing members.
- Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced **Collection Framework.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.70



The Collections Framework

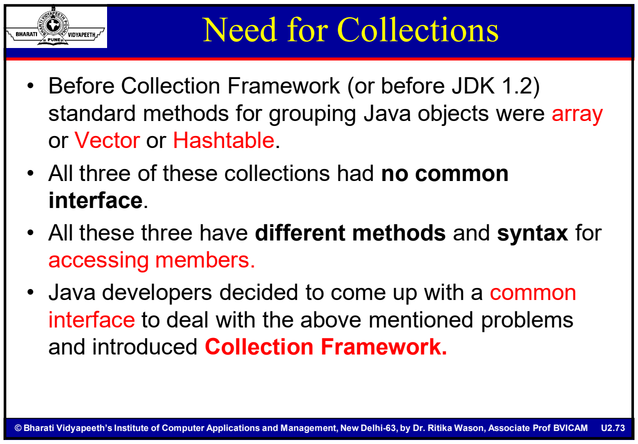
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof. BVICAM U2.71



Collections Framework

- A *collection* — sometimes called a **container** — is simply an object that groups multiple elements into a **single unit**.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collections Framework is a **sophisticated hierarchy** of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Used to store, retrieve, manipulate, and communicate aggregate data.

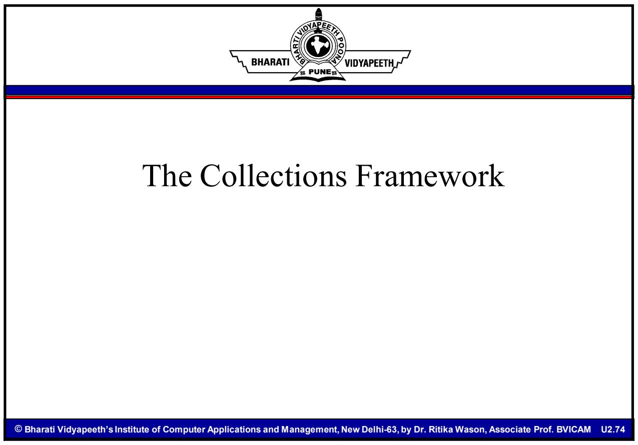
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.72



Need for Collections

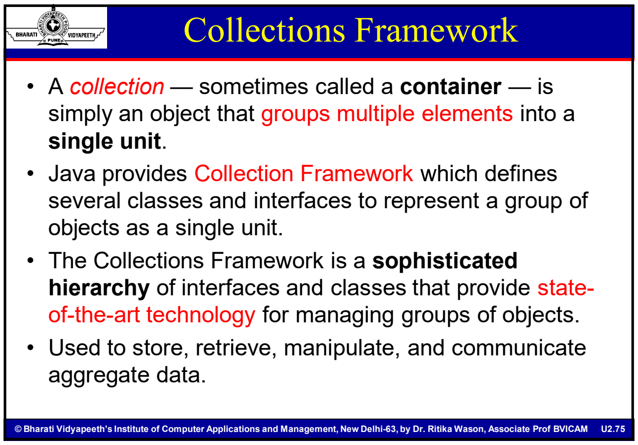
- Before Collection Framework (or before JDK 1.2) standard methods for grouping Java objects were array or Vector or Hashtable.
- All three of these collections had **no common interface**.
- All these three have **different methods** and **syntax** for accessing members.
- Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced **Collection Framework.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.73



The Collections Framework

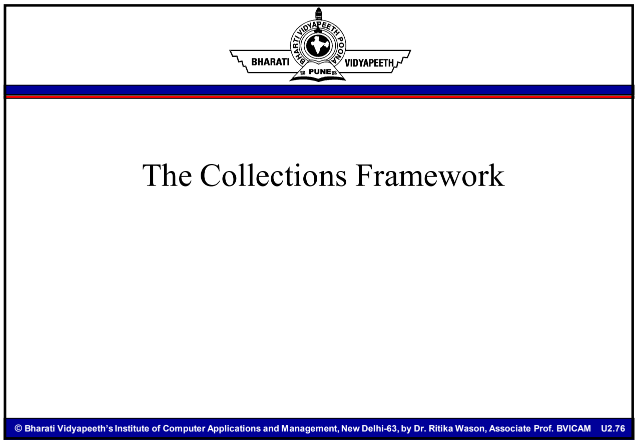
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof. BVICAM U2.74



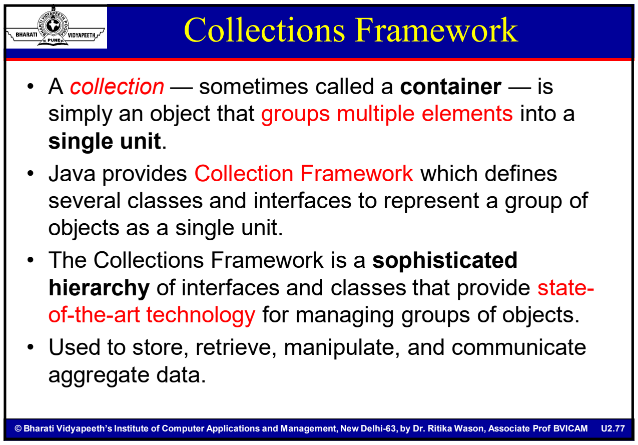
Collections Framework

- A *collection* — sometimes called a **container** — is simply an object that groups multiple elements into a **single unit**.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collections Framework is a **sophisticated hierarchy** of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Used to store, retrieve, manipulate, and communicate aggregate data.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.75

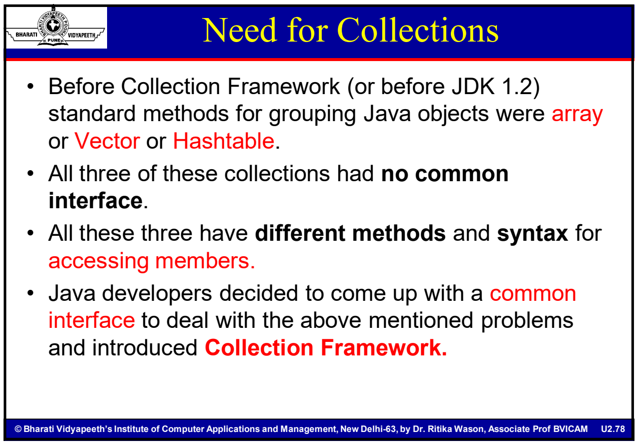


The Collections Framework



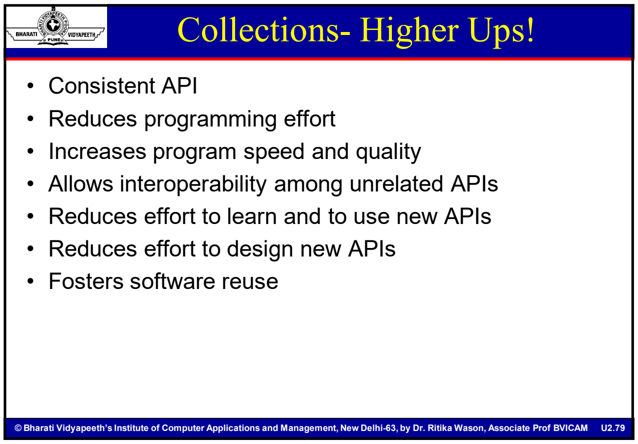
Collections Framework

- A *collection* — sometimes called a **container** — is simply an object that groups multiple elements into a **single unit**.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collections Framework is a **sophisticated hierarchy** of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Used to store, retrieve, manipulate, and communicate aggregate data.



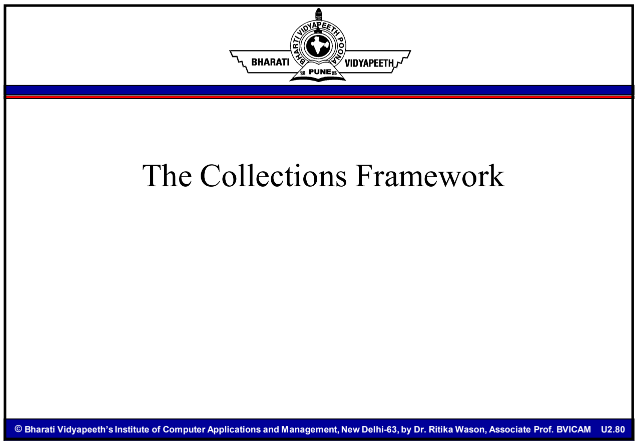
Need for Collections

- Before Collection Framework (or before JDK 1.2) standard methods for grouping Java objects were array or Vector or Hashtable.
- All three of these collections had **no common interface**.
- All these three have **different methods** and **syntax** for accessing members.
- Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced **Collection Framework.**

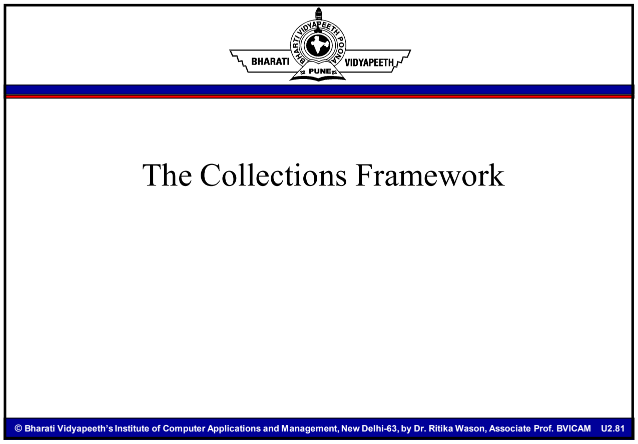


Collections- Higher Ups!

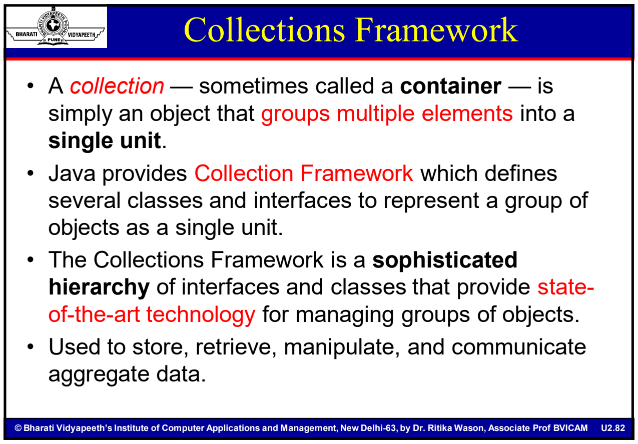
- Consistent API
- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse



The Collections Framework

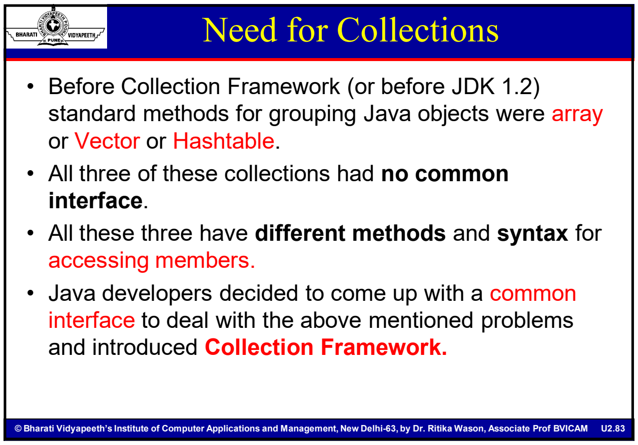


The Collections Framework



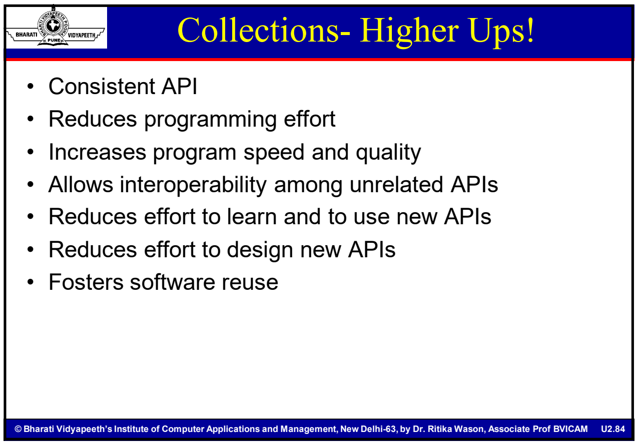
Collections Framework

- A *collection* — sometimes called a **container** — is simply an object that groups multiple elements into a **single unit**.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collections Framework is a **sophisticated hierarchy** of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Used to store, retrieve, manipulate, and communicate aggregate data.



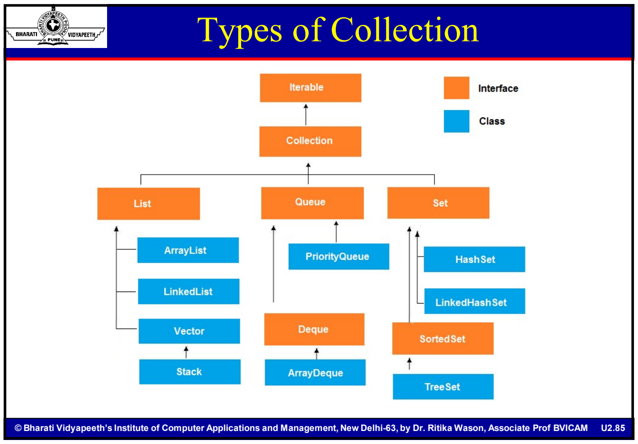
Need for Collections

- Before Collection Framework (or before JDK 1.2) standard methods for grouping Java objects were array or Vector or Hashtable.
- All three of these collections had **no common interface**.
- All these three have **different methods** and **syntax** for accessing members.
- Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced **Collection Framework.**



Collections- Higher Ups!

- Consistent API
- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse



Types of Collection

Interface

Class

Iterable

Collection

List

Queue

Set

ArrayList

LinkedList

Vector

Stack

PriorityQueue

Deque

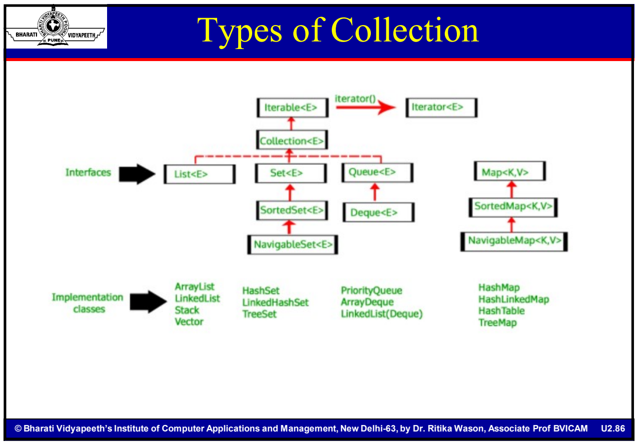
ArrayDeque

HashSet

LinkedHashSet

SortedSet

TreeSet



Types of Collection

Iterable<E>

iterator()

Iterator<E>

Collection<E>

Interfaces

List<E>

Set<E>

Queue<E>

Map<K,V>

SortedSet<E>

Deque<E>

SortedMap<K,V>

NavigableSet<E>

NavigableMap<K,V>

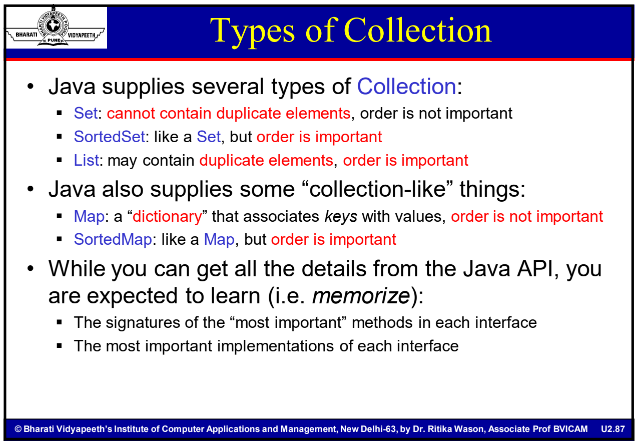
Implementation classes

ArrayList
LinkedList
Stack
Vector

HashSet
LinkedHashSet
TreeSet

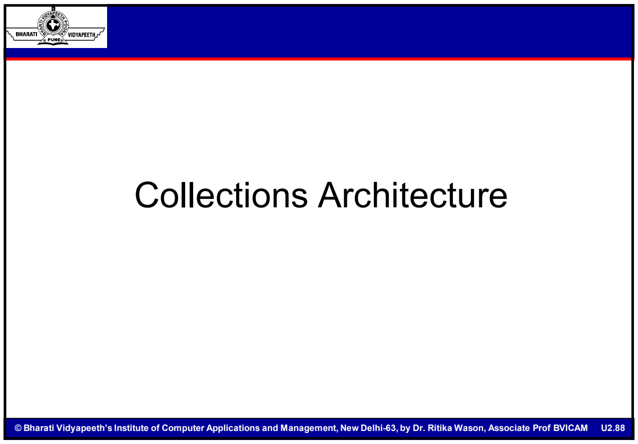
PriorityQueue
ArrayDeque
LinkedList(Deque)

HashMap
HashLinkedMap
HashTable
TreeMap

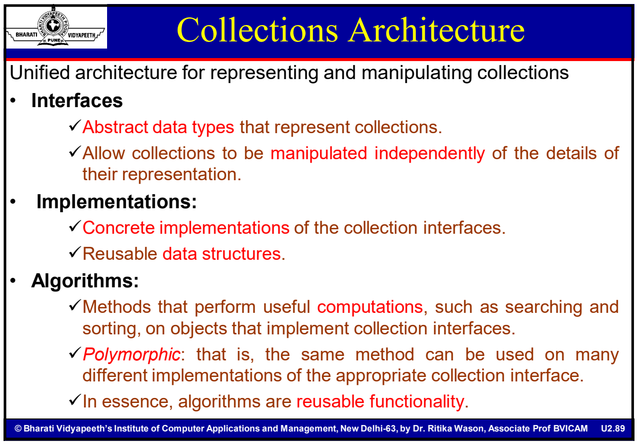


Types of Collection

- Java supplies several types of Collection:
  - Set: cannot contain duplicate elements, order is not important
  - SortedSet: like a Set, but order is important
  - List: may contain duplicate elements, order is important
- Java also supplies some "collection-like" things:
  - Map: a "dictionary" that associates *keys* with values, order is not important
  - SortedMap: like a Map, but order is important
- While you can get all the details from the Java API, you are expected to learn (i.e. *memorize*):
  - The signatures of the "most important" methods in each interface
  - The most important implementations of each interface



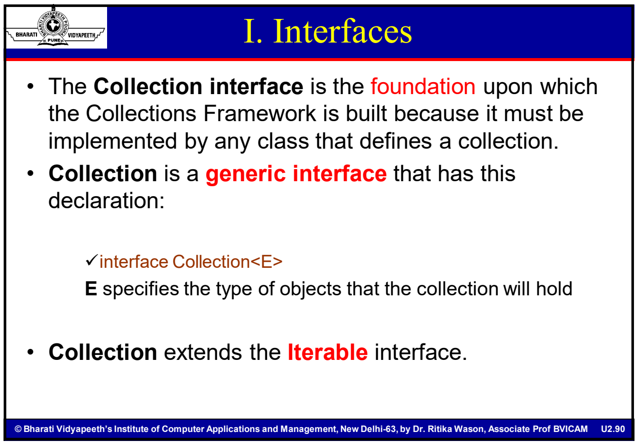
Collections Architecture



Collections Architecture

Unified architecture for representing and manipulating collections

- **Interfaces**
  - ✓Abstract data types that represent collections.
  - ✓Allow collections to be manipulated independently of the details of their representation.
- **Implementations:**
  - ✓Concrete implementations of the collection interfaces.
  - ✓Reusable data structures.
- **Algorithms:**
  - ✓Methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
  - ✓*Polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface.
  - ✓In essence, algorithms are reusable functionality.



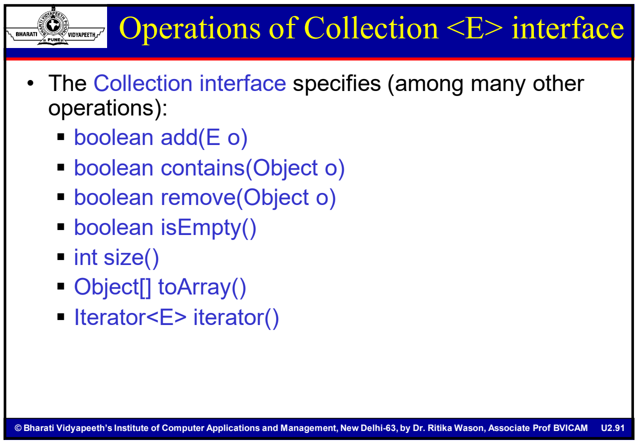
I. Interfaces

- The **Collection interface** is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.
- **Collection** is a **generic interface** that has this declaration:

  ✓interface Collection<E>

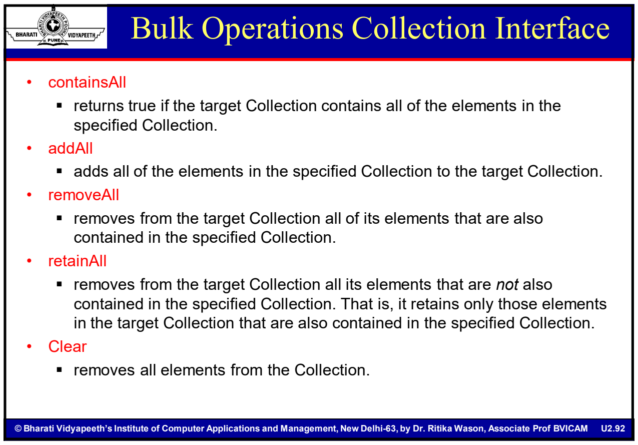
  E specifies the type of objects that the collection will hold

- **Collection** extends the **Iterable** interface.



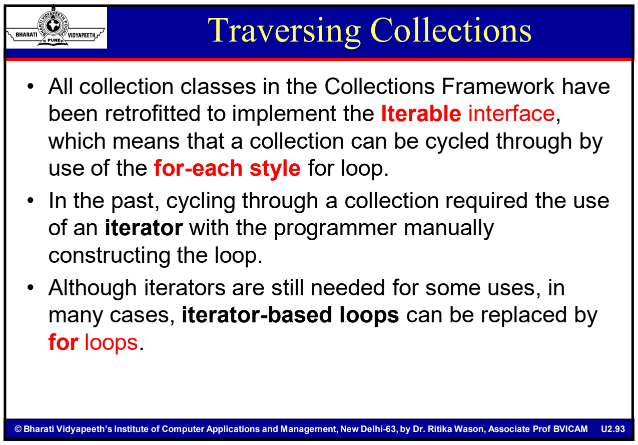
Operations of Collection <E> interface

- The Collection interface specifies (among many other operations):
  - boolean add(E o)
  - boolean contains(Object o)
  - boolean remove(Object o)
  - boolean isEmpty()
  - int size()
  - Object[] toArray()
  - Iterator<E> iterator()



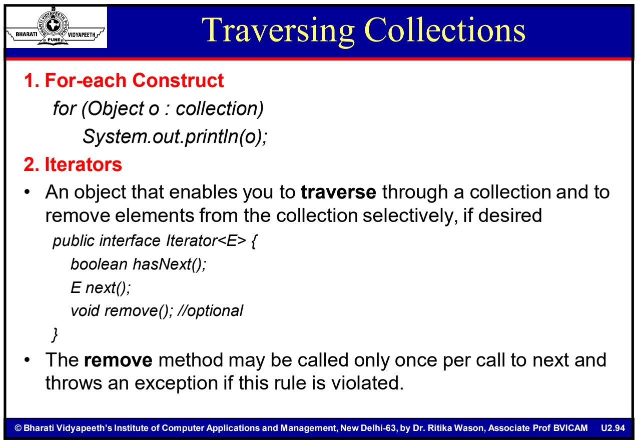
Bulk Operations Collection Interface

- 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.



Traversing Collections

- All collection classes in the Collections Framework have been retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the **for-each style** for loop.
- In the past, cycling through a collection required the use of an **iterator** with the programmer manually constructing the loop.
- Although iterators are still needed for some uses, in many cases, **iterator-based loops** can be replaced by **for** loops.



Traversing Collections

1. For-each Construct

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

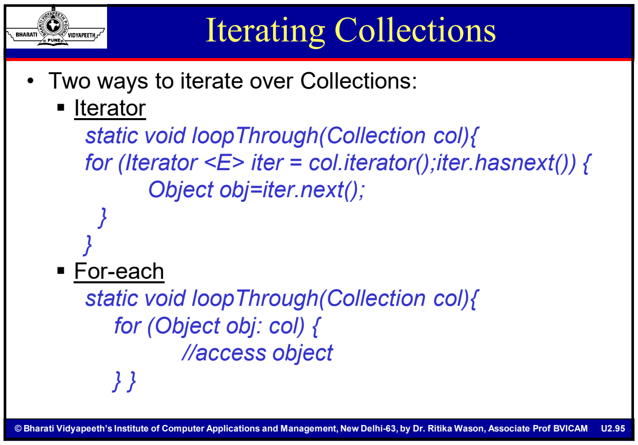
2. Iterators

- An object that enables you to **traverse** through a collection and to remove elements from the collection selectively, if desired

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- The **remove** method may be called only once per call to next and throws an exception if this rule is violated.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.94



Iterating Collections

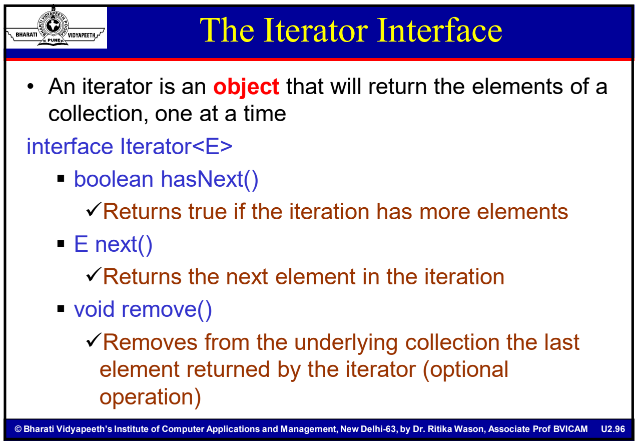
- Two ways to iterate over Collections:
  - Iterator

```
static void loopThrough(Collection col){
for (Iterator <E> iter = col.iterator();iter.hasnext()) {
        Object obj=iter.next();
  }
}
```

  - For-each

```
static void loopThrough(Collection col){
    for (Object obj: col) {
             //access object
    } }
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.95



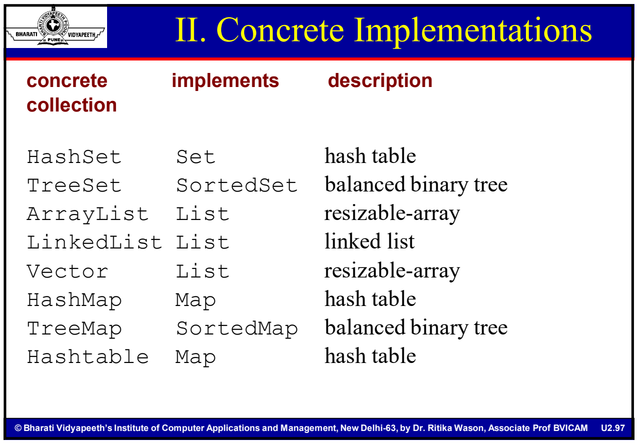
The Iterator Interface

- An iterator is an **object** that will return the elements of a collection, one at a time

interface Iterator<E>

- boolean hasNext()
  - ✓Returns true if the iteration has more elements
- E next()
  - ✓Returns the next element in the iteration
- void remove()
  - ✓Removes from the underlying collection the last element returned by the iterator (optional operation)

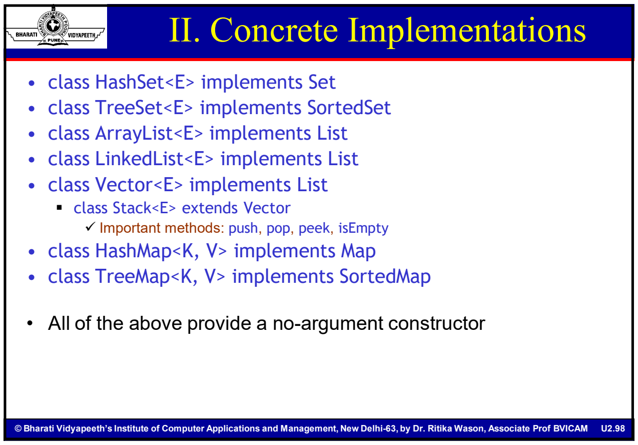
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.96



II. Concrete Implementations

concrete collection	implements	description
`HashSet`	`Set`	hash table
`TreeSet`	`SortedSet`	balanced binary tree
`ArrayList`	`List`	resizable-array
`LinkedList`	`List`	linked list
`Vector`	`List`	resizable-array
`HashMap`	`Map`	hash table
`TreeMap`	`SortedMap`	balanced binary tree
`Hashtable`	`Map`	hash table

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.97

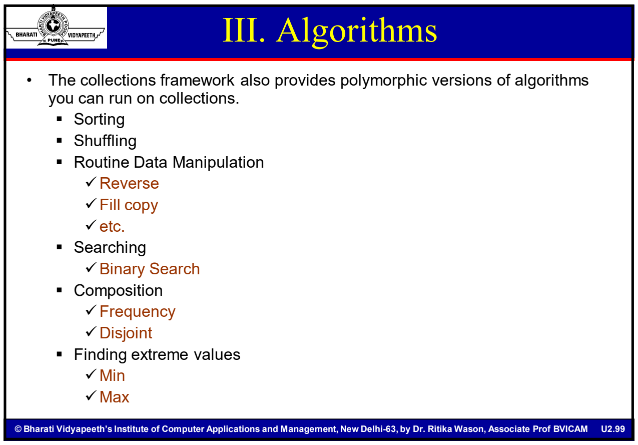


II. Concrete Implementations

- class HashSet<E> implements Set
- class TreeSet<E> implements SortedSet
- class ArrayList<E> implements List
- class LinkedList<E> implements List
- class Vector<E> implements List
  - class Stack<E> extends Vector
    - ✓ Important methods: push, pop, peek, isEmpty
- class HashMap<K, V> implements Map
- class TreeMap<K, V> implements SortedMap

- All of the above provide a no-argument constructor

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.98

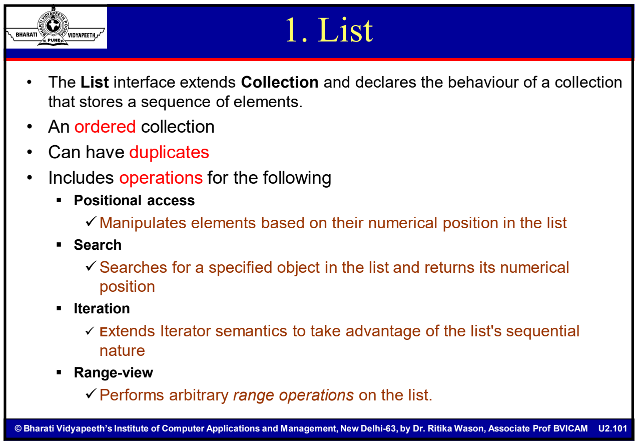


III. Algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
    - ✓ Reverse
    - ✓ Fill copy
    - ✓ etc.
  - Searching
    - ✓ Binary Search
  - Composition
    - ✓ Frequency
    - ✓ Disjoint
  - Finding extreme values
    - ✓ Min
    - ✓ Max

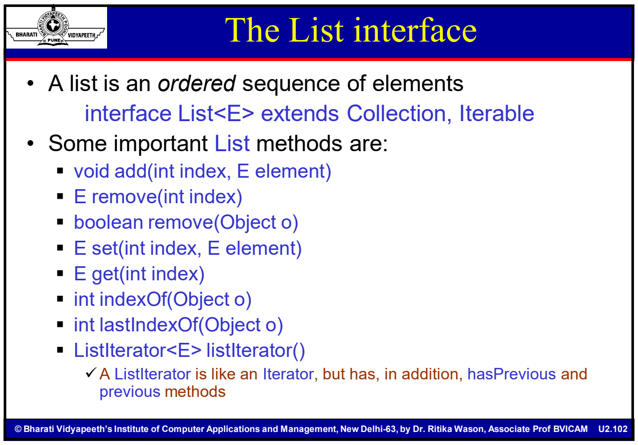
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.99

Collection Implementations



1. List

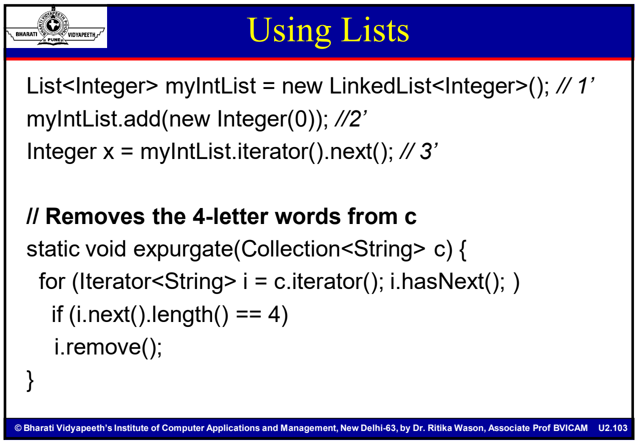
- The **List** interface extends **Collection** and declares the behaviour of a collection that stores a sequence of elements.
- An ordered collection
- Can have duplicates
- Includes operations for the following
  - **Positional access**
    - ✓ Manipulates elements based on their numerical position in the list
  - **Search**
    - ✓ Searches for a specified object in the list and returns its numerical position
  - **Iteration**
    - ✓ Extends Iterator semantics to take advantage of the list's sequential nature
  - **Range-view**
    - ✓ Performs arbitrary *range operations* on the list.



The List interface

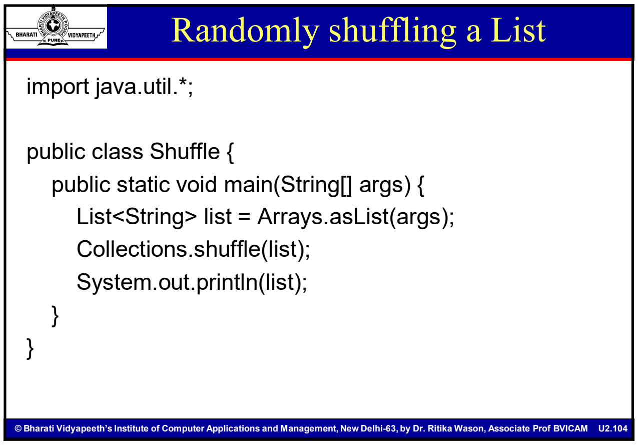
- A list is an *ordered* sequence of elements

  interface List<E> extends Collection, Iterable
- Some important List methods are:
  - void add(int index, E element)
  - E remove(int index)
  - boolean remove(Object o)
  - E set(int index, E element)
  - E get(int index)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)
  - ListIterator<E> listIterator()
    - ✓ A ListIterator is like an Iterator, but has, in addition, hasPrevious and previous methods



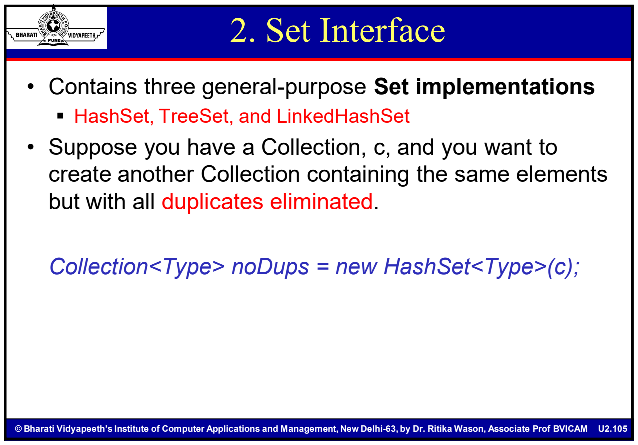
Using Lists

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'

// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
  for (Iterator<String> i = c.iterator(); i.hasNext(); )
    if (i.next().length() == 4)
     i.remove();
}
```



Randomly shuffling a List

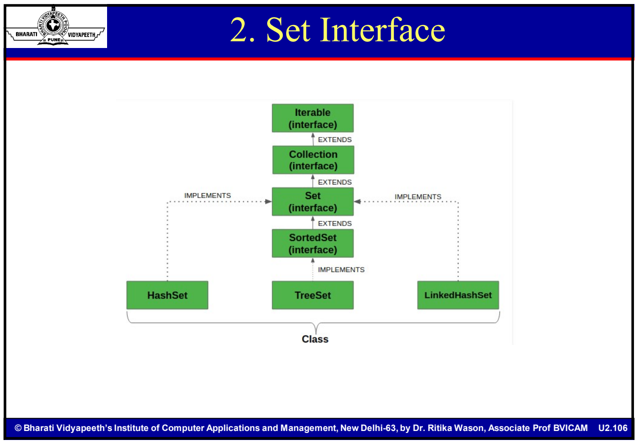
```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```



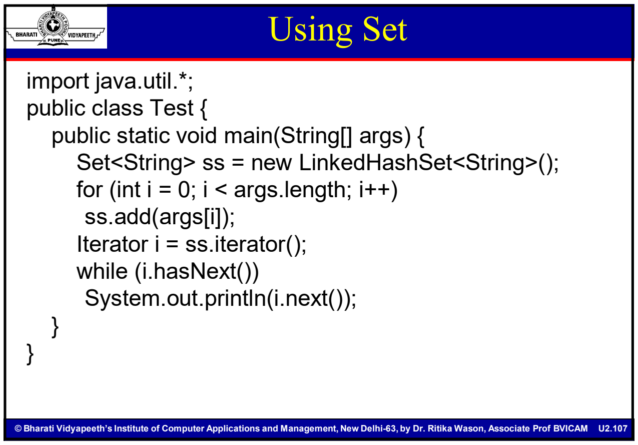
2. Set Interface

- Contains three general-purpose **Set implementations**
  - HashSet, TreeSet, and LinkedHashSet
- Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated.

Collection<Type> noDups = new HashSet<Type>(c);

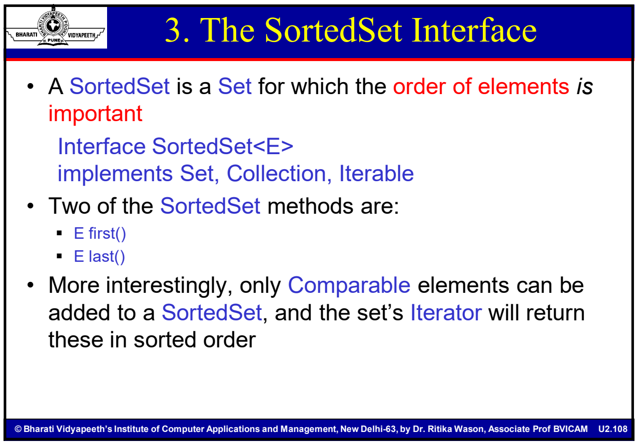


2. Set Interface



Using Set

```
import java.util.*;
public class Test {
   public static void main(String[] args) {
      Set<String> ss = new LinkedHashSet<String>();
      for (int i = 0; i < args.length; i++)
       ss.add(args[i]);
      Iterator i = ss.iterator();
      while (i.hasNext())
       System.out.println(i.next());
   }
}
```

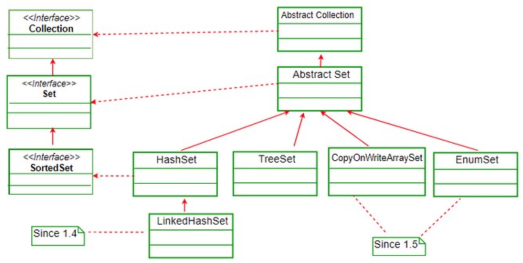


3. The SortedSet Interface

- A SortedSet is a Set for which the order of elements *is* important

  Interface SortedSet<E>
  implements Set, Collection, Iterable
- Two of the SortedSet methods are:
  - E first()
  - E last()
- More interestingly, only Comparable elements can be added to a SortedSet, and the set's Iterator will return these in sorted order

3. The SortedSet Interface



4. The HashSet

- **HashSet** extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

  ✓class HashSet<E>

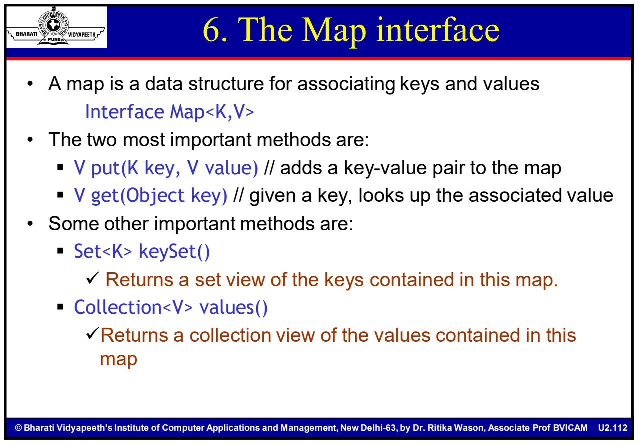
  Here, **E** specifies the type of objects that the set will hold.
- **HashSet** does not guarantee the order of its elements
- For sorted storage **TreeSet**, is a better choice.

5. The TreeSet

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.

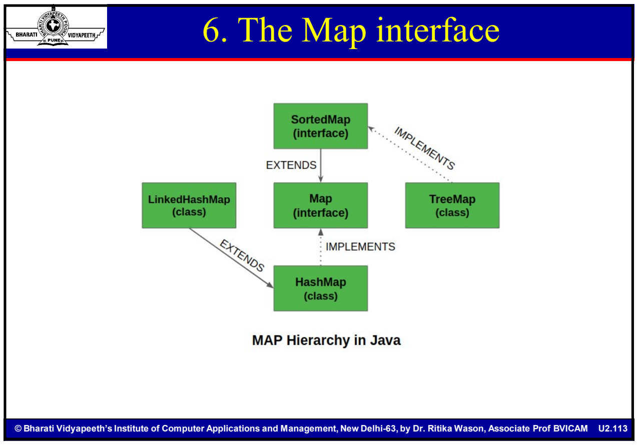
  ✓class TreeSet<E>

  Here, **E** specifies the type of objects that the set will hold
- It creates a collection that uses a tree for storage.
- Objects are stored in sorted, ascending order
- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of **sorted information** that must be found quickly.



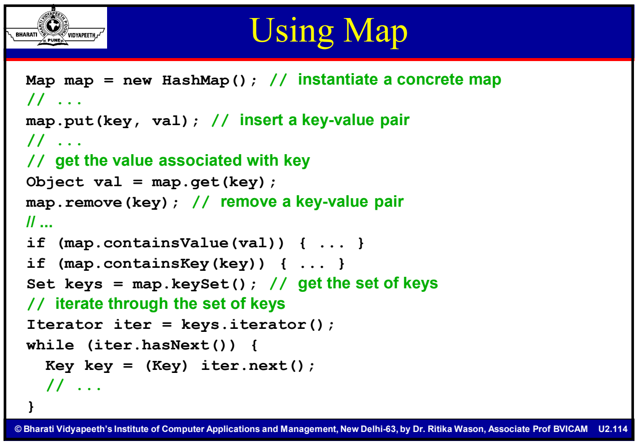
6. The Map interface

- A map is a data structure for associating keys and values
  Interface Map<K,V>
- The two most important methods are:
  - V put(K key, V value) // adds a key-value pair to the map
  - V get(Object key) // given a key, looks up the associated value
- Some other important methods are:
  - Set<K> keySet()
    - ✓ Returns a set view of the keys contained in this map.
  - Collection<V> values()
    - ✓Returns a collection view of the values contained in this map



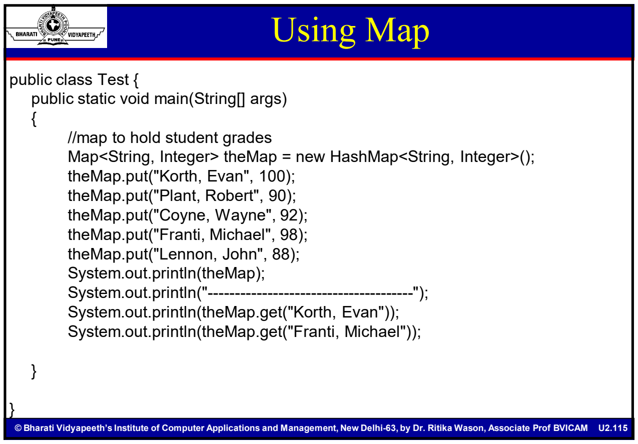
6. The Map interface

MAP Hierarchy in Java



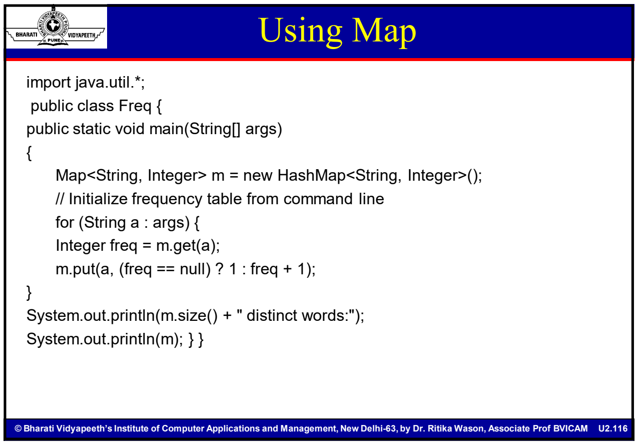
Using Map

```
Map map = new HashMap(); // instantiate a concrete map
// ...
map.put(key, val); // insert a key-value pair
// ...
// get the value associated with key
Object val = map.get(key);
map.remove(key); // remove a key-value pair
// ...
if (map.containsValue(val)) { ... }
if (map.containsKey(key)) { ... }
Set keys = map.keySet(); // get the set of keys
// iterate through the set of keys
Iterator iter = keys.iterator();
while (iter.hasNext()) {
  Key key = (Key) iter.next();
  // ...
}
```



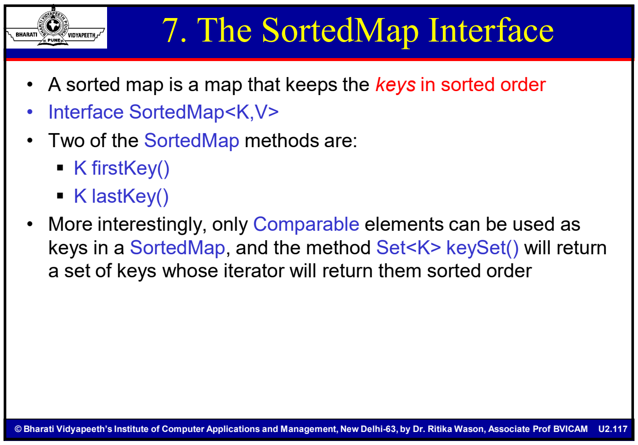
Using Map

```
public class Test {
    public static void main(String[] args)
    {
          //map to hold student grades
          Map<String, Integer> theMap = new HashMap<String, Integer>();
          theMap.put("Korth, Evan", 100);
          theMap.put("Plant, Robert", 90);
          theMap.put("Coyne, Wayne", 92);
          theMap.put("Franti, Michael", 98);
          theMap.put("Lennon, John", 88);
          System.out.println(theMap);
          System.out.println("--------------------------------------");
          System.out.println(theMap.get("Korth, Evan"));
          System.out.println(theMap.get("Franti, Michael"));

    }
}
```



Using Map

```
import java.util.*;
 public class Freq {
public static void main(String[] args)
{
     Map<String, Integer> m = new HashMap<String, Integer>();
     // Initialize frequency table from command line
     for (String a : args) {
     Integer freq = m.get(a);
     m.put(a, (freq == null) ? 1 : freq + 1);
}
System.out.println(m.size() + " distinct words:");
System.out.println(m); } }
```



7. The SortedMap Interface

- A sorted map is a map that keeps the *keys* in sorted order
- Interface SortedMap<K,V>
- Two of the SortedMap methods are:
  - K firstKey()
  - K lastKey()
- More interestingly, only Comparable elements can be used as keys in a SortedMap, and the method Set<K> keySet() will return a set of keys whose iterator will return them sorted order

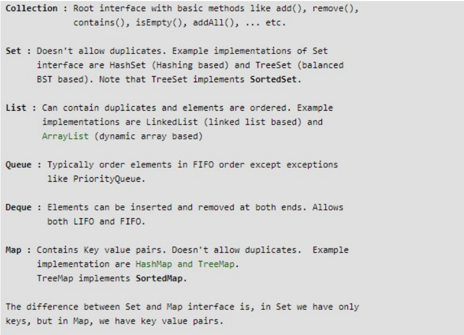
8. The HashMap

- The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.
- This allows the execution time of **get( )** and **put( )** to remain constant even for large sets.
- **HashMap** is a generic class that has this declaration:
  ✓class HashMap<K, V>
  Here, **K** specifies the type of keys, and **V** specifies the type of values.

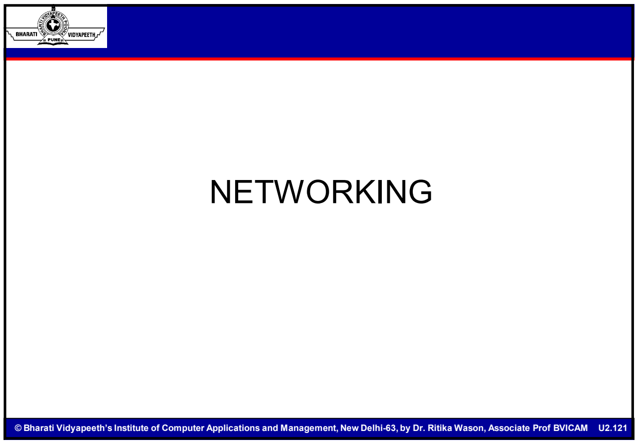
9. Properties Collection

- **Properties** is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**.
- Used in the type of object returned by System.getProperties( ) when obtaining **environmental values**
- Used to store **configurations**.

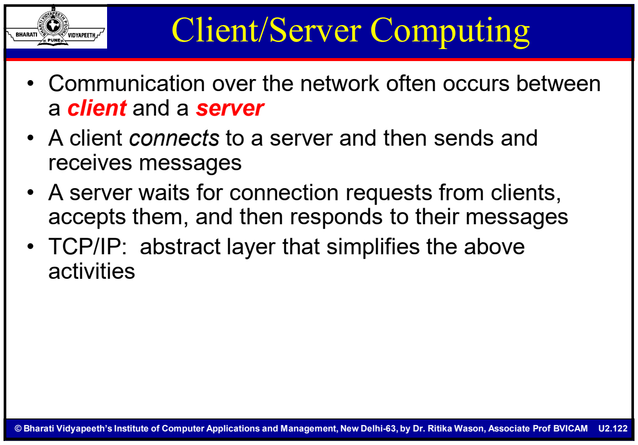
Collections at a glance…



```
Collection : Root interface with basic methods like add(), remove(),
             contains(), isEmpty(), addAll(), ... etc.

Set : Doesn't allow duplicates. Example implementations of Set
      interface are HashSet (Hashing based) and TreeSet (balanced
      BST based). Note that TreeSet implements SortedSet.

List : Can contain duplicates and elements are ordered. Example
       implementations are LinkedList (linked list based) and
       ArrayList (dynamic array based)

Queue : Typically order elements in FIFO order except exceptions
        like PriorityQueue.

Deque : Elements can be inserted and removed at both ends. Allows
        both LIFO and FIFO.

Map : Contains Key value pairs. Doesn't allow duplicates.  Example
      implementation are HashMap and TreeMap.
      TreeMap implements SortedMap.

The difference between Set and Map interface is, in Set we have only
keys, but in Map, we have key value pairs.
```

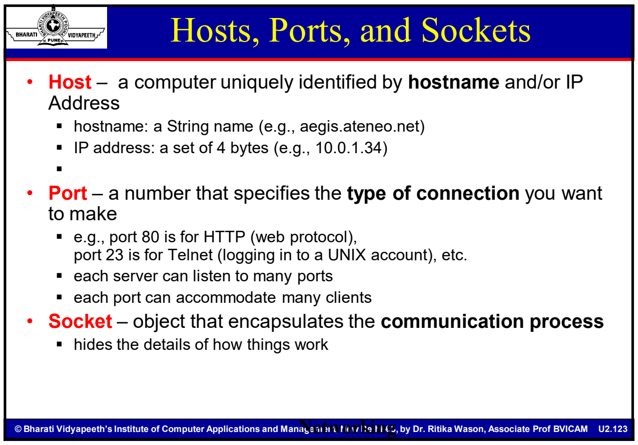


NETWORKING



Client/Server Computing

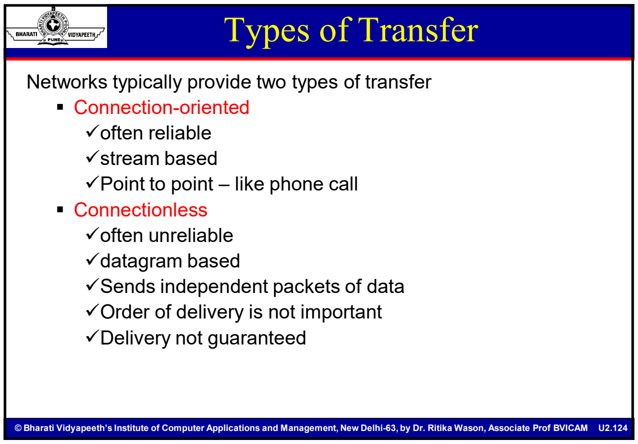
- Communication over the network often occurs between a ***client*** and a ***server***
- A client *connects* to a server and then sends and receives messages
- A server waits for connection requests from clients, accepts them, and then responds to their messages
- TCP/IP: abstract layer that simplifies the above activities



Hosts, Ports, and Sockets

- **Host** – a computer uniquely identified by **hostname** and/or IP Address
  - hostname: a String name (e.g., aegis.ateneo.net)
  - IP address: a set of 4 bytes (e.g., 10.0.1.34)
  -
- **Port** – a number that specifies the **type of connection** you want to make
  - e.g., port 80 is for HTTP (web protocol), port 23 is for Telnet (logging in to a UNIX account), etc.
  - each server can listen to many ports
  - each port can accommodate many clients
- **Socket** – object that encapsulates the **communication process**
  - hides the details of how things work

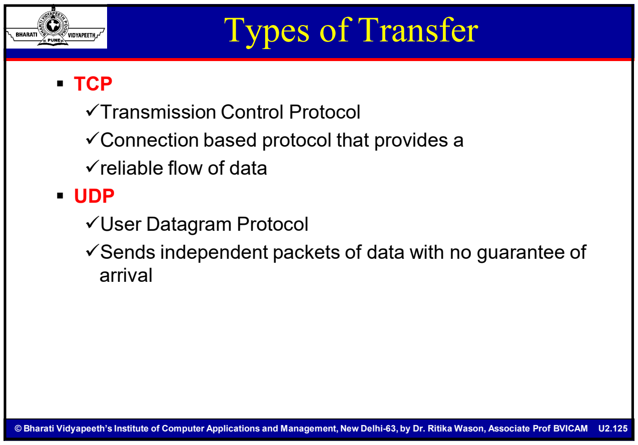
Networking



Types of Transfer

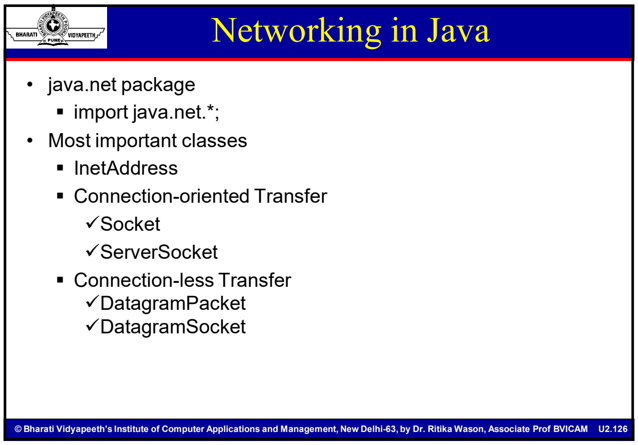
Networks typically provide two types of transfer

- Connection-oriented
  - ✓often reliable
  - ✓stream based
  - ✓Point to point – like phone call
- Connectionless
  - ✓often unreliable
  - ✓datagram based
  - ✓Sends independent packets of data
  - ✓Order of delivery is not important
  - ✓Delivery not guaranteed



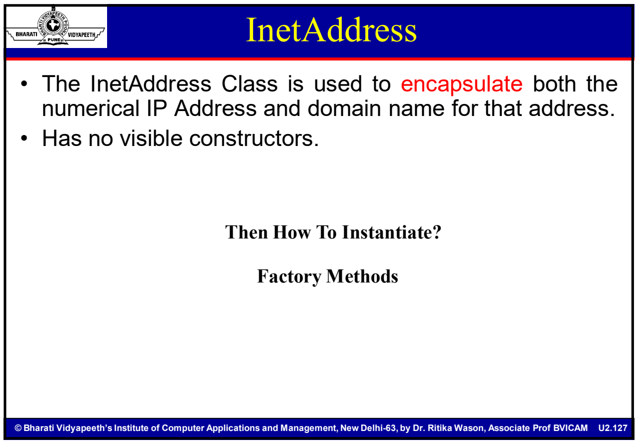
Types of Transfer

- **TCP**
  - ✓Transmission Control Protocol
  - ✓Connection based protocol that provides a
  - ✓reliable flow of data
- **UDP**
  - ✓User Datagram Protocol
  - ✓Sends independent packets of data with no guarantee of arrival



Networking in Java

- java.net package
  - import java.net.*;
- Most important classes
  - InetAddress
  - Connection-oriented Transfer
    - ✓Socket
    - ✓ServerSocket
  - Connection-less Transfer
    - ✓DatagramPacket
    - ✓DatagramSocket

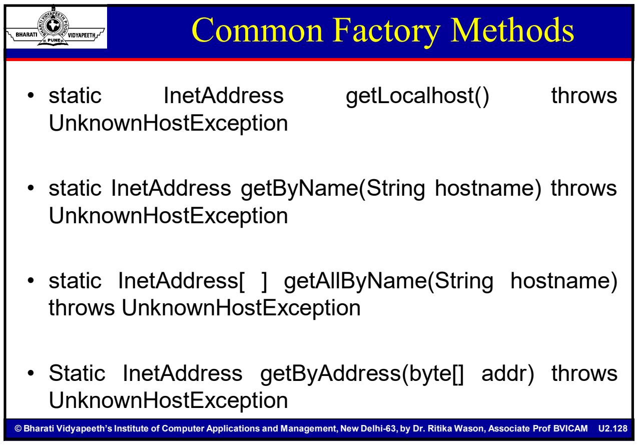


InetAddress

- The InetAddress Class is used to encapsulate both the numerical IP Address and domain name for that address.
- Has no visible constructors.

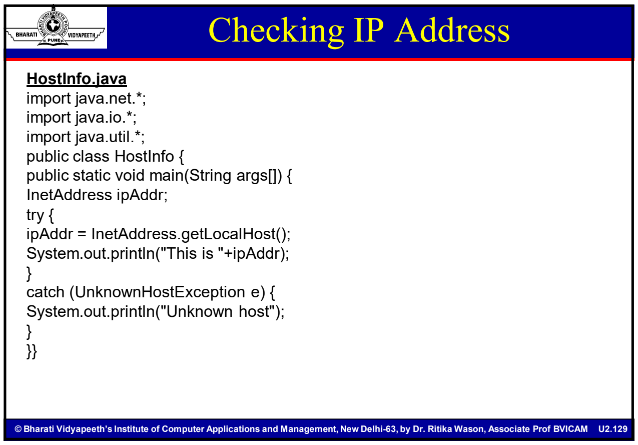
Then How To Instantiate?

Factory Methods



Common Factory Methods

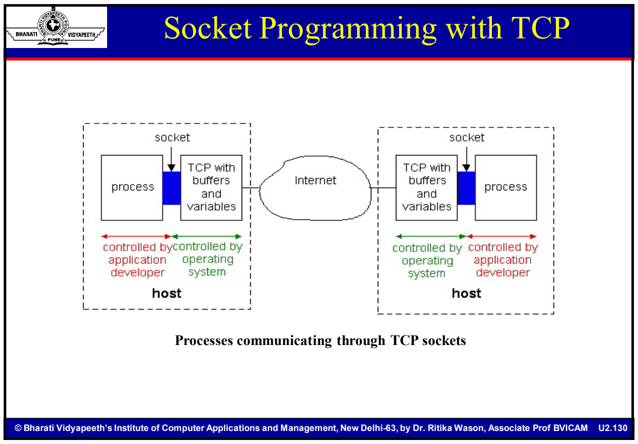
- static InetAddress getLocalhost() throws UnknownHostException
- static InetAddress getByName(String hostname) throws UnknownHostException
- static InetAddress[ ] getAllByName(String hostname) throws UnknownHostException
- Static InetAddress getByAddress(byte[] addr) throws UnknownHostException



Checking IP Address

HostInfo.java

```
import java.net.*;
import java.io.*;
import java.util.*;
public class HostInfo {
public static void main(String args[]) {
InetAddress ipAddr;
try {
ipAddr = InetAddress.getLocalHost();
System.out.println("This is "+ipAddr);
}
catch (UnknownHostException e) {
System.out.println("Unknown host");
}
}}
```



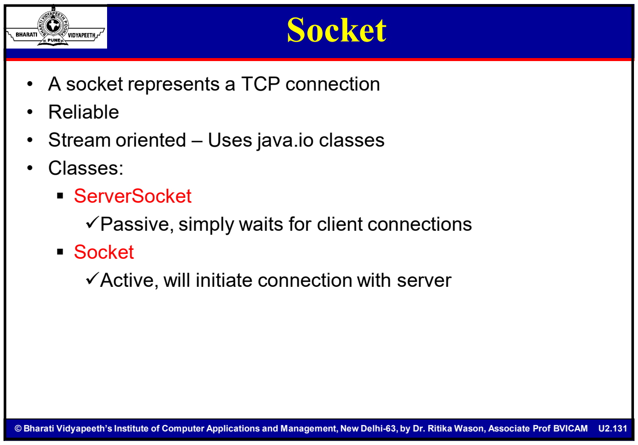
Socket Programming with TCP

socket
process
TCP with buffers and variables
controlled by application developer
controlled by operating system
host

Internet

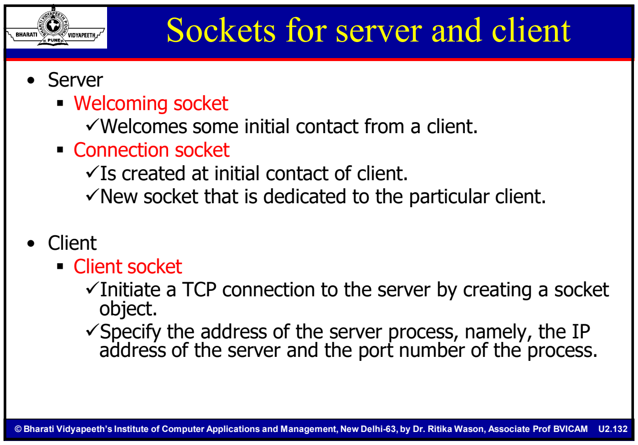
socket
TCP with buffers and variables
process
controlled by operating system
controlled by application developer
host

Processes communicating through TCP sockets



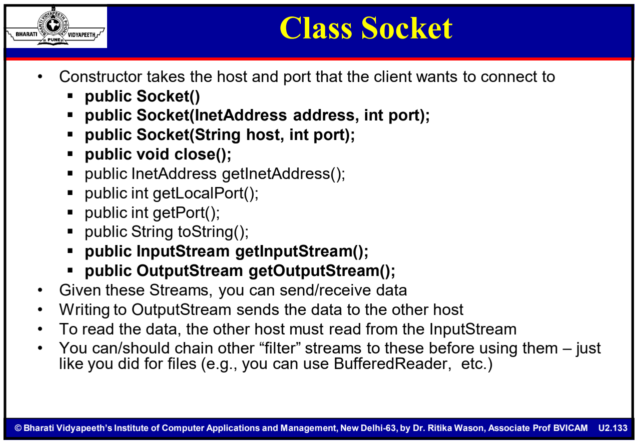
Socket

- A socket represents a TCP connection
- Reliable
- Stream oriented – Uses java.io classes
- Classes:
  - ServerSocket
    - ✓Passive, simply waits for client connections
  - Socket
    - ✓Active, will initiate connection with server



Sockets for server and client

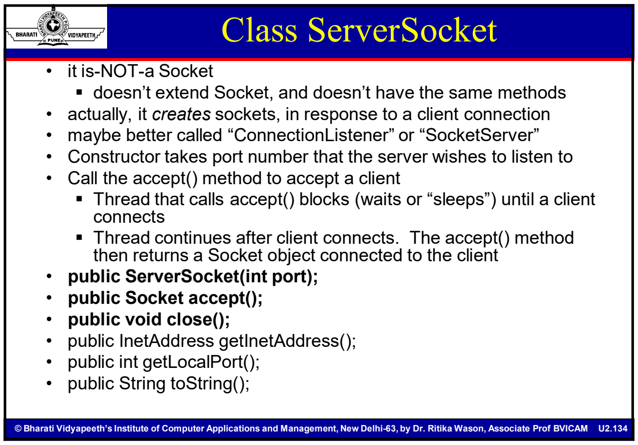
- Server
  - Welcoming socket
    - ✓Welcomes some initial contact from a client.
  - Connection socket
    - ✓Is created at initial contact of client.
    - ✓New socket that is dedicated to the particular client.
- Client
  - Client socket
    - ✓Initiate a TCP connection to the server by creating a socket object.
    - ✓Specify the address of the server process, namely, the IP address of the server and the port number of the process.



Class Socket

- Constructor takes the host and port that the client wants to connect to
  - **public Socket()**
  - **public Socket(InetAddress address, int port);**
  - **public Socket(String host, int port);**
  - **public void close();**
  - public InetAddress getInetAddress();
  - public int getLocalPort();
  - public int getPort();
  - public String toString();
  - **public InputStream getInputStream();**
  - **public OutputStream getOutputStream();**
- Given these Streams, you can send/receive data
- Writing to OutputStream sends the data to the other host
- To read the data, the other host must read from the InputStream
- You can/should chain other “filter” streams to these before using them – just like you did for files (e.g., you can use BufferedReader, etc.)

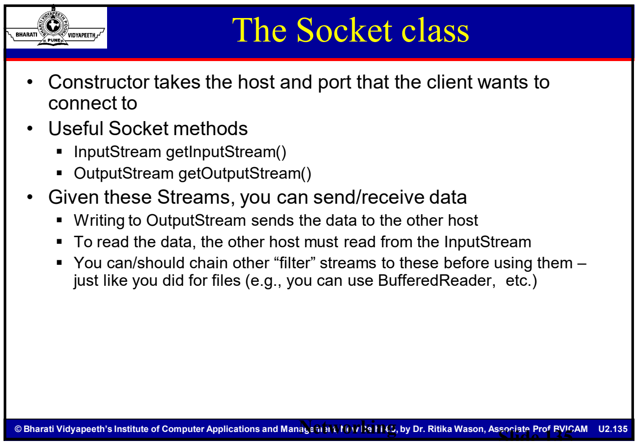
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.133



Class ServerSocket

- it is-NOT-a Socket
  - doesn’t extend Socket, and doesn’t have the same methods
- actually, it *creates* sockets, in response to a client connection
- maybe better called “ConnectionListener” or “SocketServer”
- Constructor takes port number that the server wishes to listen to
- Call the accept() method to accept a client
  - Thread that calls accept() blocks (waits or “sleeps”) until a client connects
  - Thread continues after client connects. The accept() method then returns a Socket object connected to the client
- **public ServerSocket(int port);**
- **public Socket accept();**
- **public void close();**
- public InetAddress getInetAddress();
- public int getLocalPort();
- public String toString();

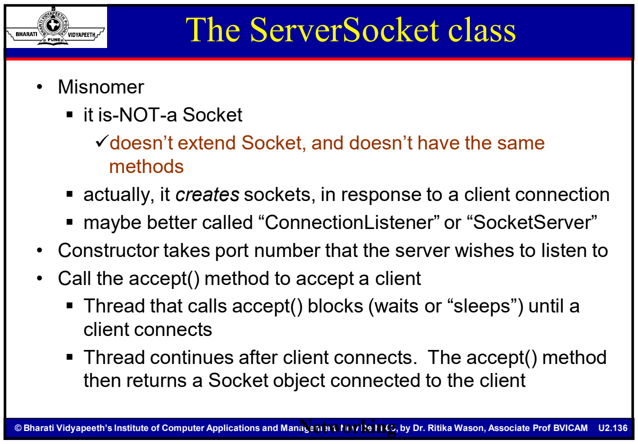
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.134



The Socket class

- Constructor takes the host and port that the client wants to connect to
- Useful Socket methods
  - InputStream getInputStream()
  - OutputStream getOutputStream()
- Given these Streams, you can send/receive data
  - Writing to OutputStream sends the data to the other host
  - To read the data, the other host must read from the InputStream
  - You can/should chain other “filter” streams to these before using them – just like you did for files (e.g., you can use BufferedReader, etc.)

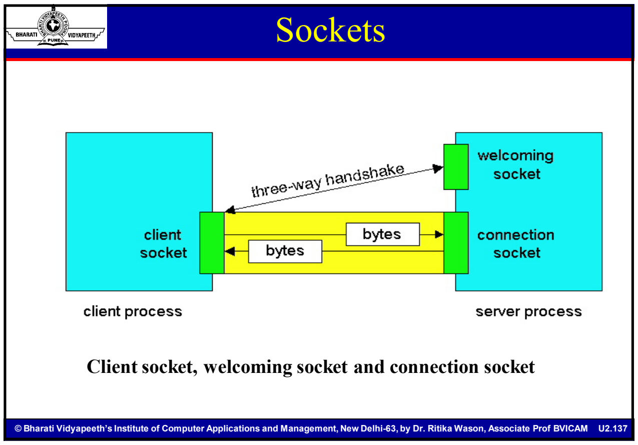
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.135



The ServerSocket class

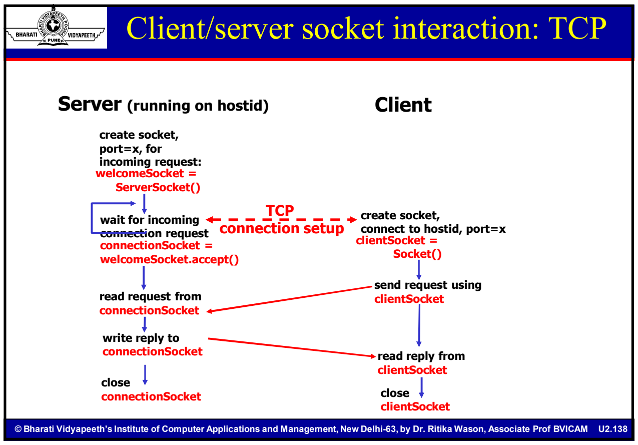
- Misnomer
  - it is-NOT-a Socket
    - ✓doesn’t extend Socket, and doesn’t have the same methods
  - actually, it *creates* sockets, in response to a client connection
  - maybe better called “ConnectionListener” or “SocketServer”
- Constructor takes port number that the server wishes to listen to
- Call the accept() method to accept a client
  - Thread that calls accept() blocks (waits or “sleeps”) until a client connects
  - Thread continues after client connects. The accept() method then returns a Socket object connected to the client

Networking



Sockets

Client socket, welcoming socket and connection socket



Client/server socket interaction: TCP

Server (running on hostid)

create socket, port=x, for incoming request: welcomeSocket = ServerSocket()

wait for incoming connection request connectionSocket = welcomeSocket.accept()

read request from connectionSocket

write reply to connectionSocket

close connectionSocket

Client

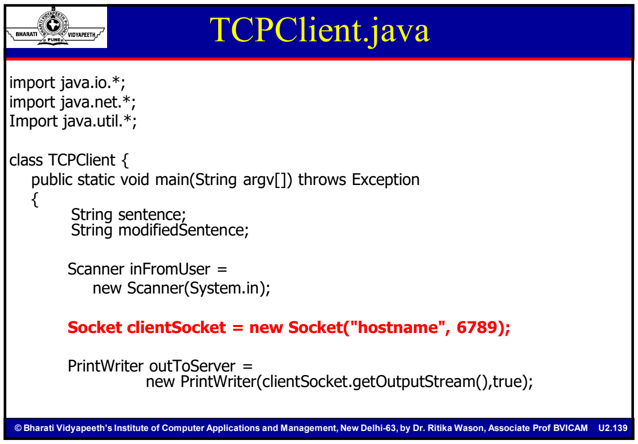
create socket, connect to hostid, port=x clientSocket = Socket()

send request using clientSocket

read reply from clientSocket

close clientSocket

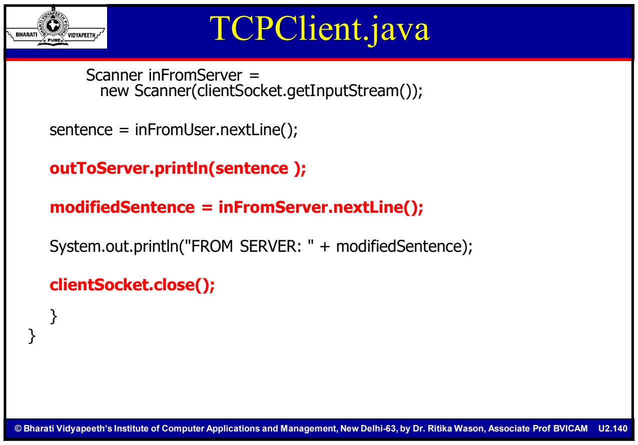
TCP connection setup



TCPClient.java

```
import java.io.*;
import java.net.*;
Import java.util.*;

class TCPClient {
    public static void main(String argv[]) throws Exception
    {
           String sentence;
           String modifiedSentence;

          Scanner inFromUser =
              new Scanner(System.in);

          Socket clientSocket = new Socket("hostname", 6789);

          PrintWriter outToServer =
                        new PrintWriter(clientSocket.getOutputStream(),true);
```

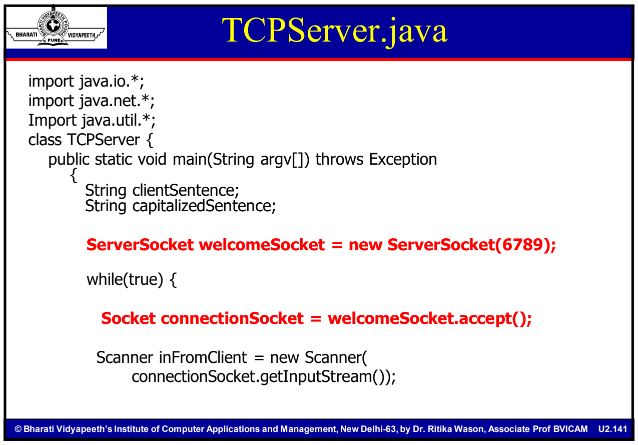
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.139



TCPClient.java

```
             Scanner inFromServer =
               new Scanner(clientSocket.getInputStream());

        sentence = inFromUser.nextLine();

        outToServer.println(sentence );

        modifiedSentence = inFromServer.nextLine();

        System.out.println("FROM SERVER: " + modifiedSentence);

        clientSocket.close();

        }
    }
```

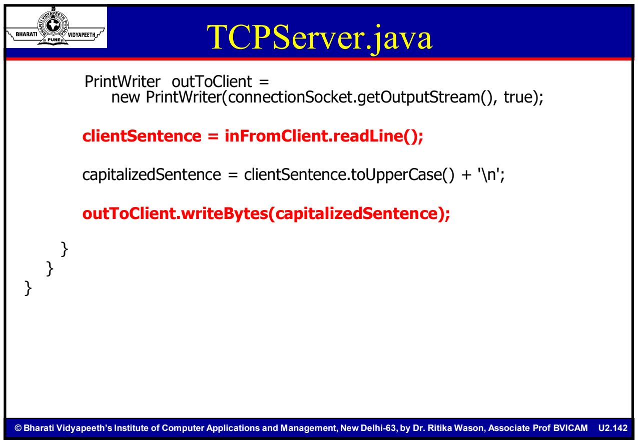
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.140



TCPServer.java

```
import java.io.*;
import java.net.*;
Import java.util.*;
class TCPServer {
    public static void main(String argv[]) throws Exception
        {
          String clientSentence;
          String capitalizedSentence;

          ServerSocket welcomeSocket = new ServerSocket(6789);

          while(true) {

            Socket connectionSocket = welcomeSocket.accept();

           Scanner inFromClient = new Scanner(
                  connectionSocket.getInputStream());
```

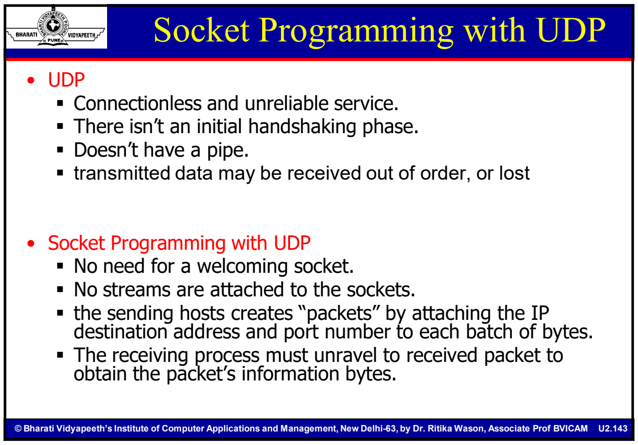
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.141



TCPServer.java

```
        PrintWriter  outToClient =
           new PrintWriter(connectionSocket.getOutputStream(), true);

        clientSentence = inFromClient.readLine();

        capitalizedSentence = clientSentence.toUpperCase() + '\n';

        outToClient.writeBytes(capitalizedSentence);

      }
    }
  }
```

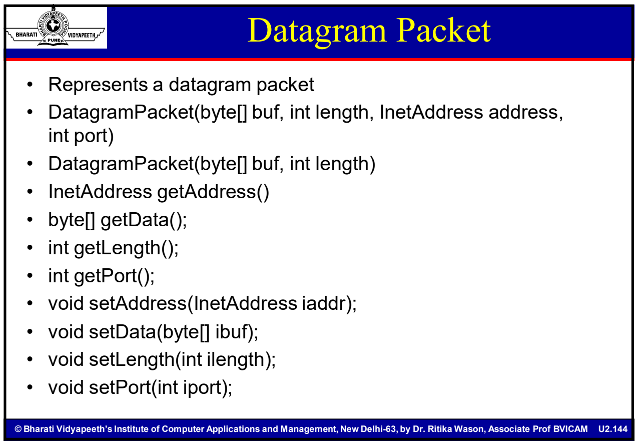
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.142



Socket Programming with UDP

- UDP
  - Connectionless and unreliable service.
  - There isn't an initial handshaking phase.
  - Doesn't have a pipe.
  - transmitted data may be received out of order, or lost
- Socket Programming with UDP
  - No need for a welcoming socket.
  - No streams are attached to the sockets.
  - the sending hosts creates "packets" by attaching the IP destination address and port number to each batch of bytes.
  - The receiving process must unravel to received packet to obtain the packet's information bytes.

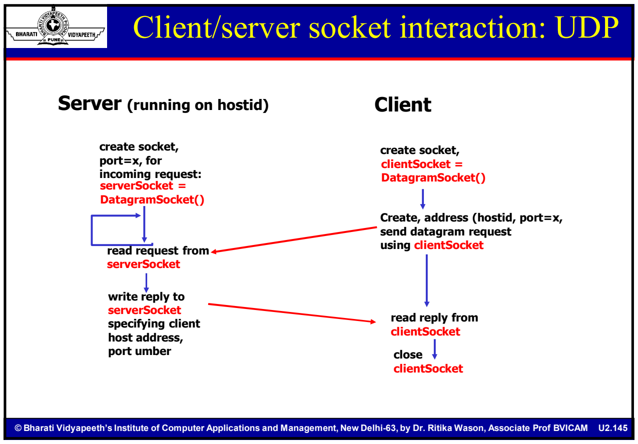
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.143



Datagram Packet

- Represents a datagram packet
- DatagramPacket(byte[] buf, int length, InetAddress address, int port)
- DatagramPacket(byte[] buf, int length)
- InetAddress getAddress()
- byte[] getData();
- int getLength();
- int getPort();
- void setAddress(InetAddress iaddr);
- void setData(byte[] ibuf);
- void setLength(int ilength);
- void setPort(int iport);

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.144



Client/server socket interaction: UDP

Server (running on hostid)

create socket, port=x, for incoming request: serverSocket = DatagramSocket()

read request from serverSocket

write reply to serverSocket specifying client host address, port umber

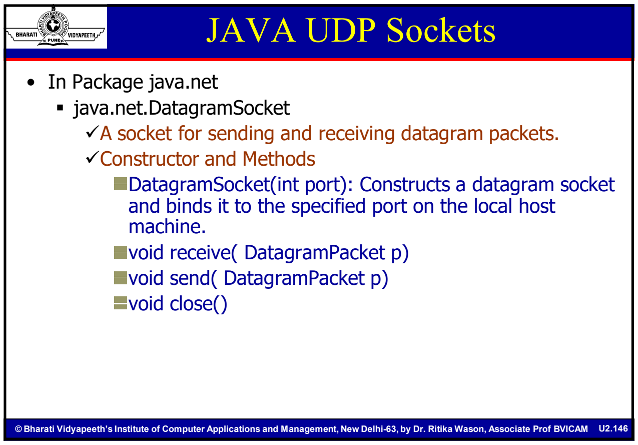
Client

create socket, clientSocket = DatagramSocket()

Create, address (hostid, port=x, send datagram request using clientSocket

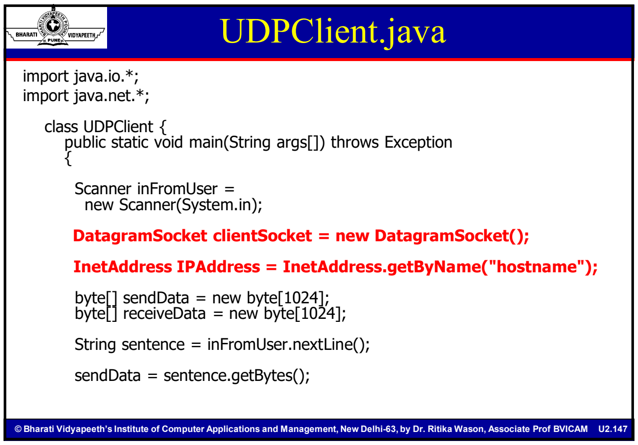
read reply from clientSocket

close clientSocket



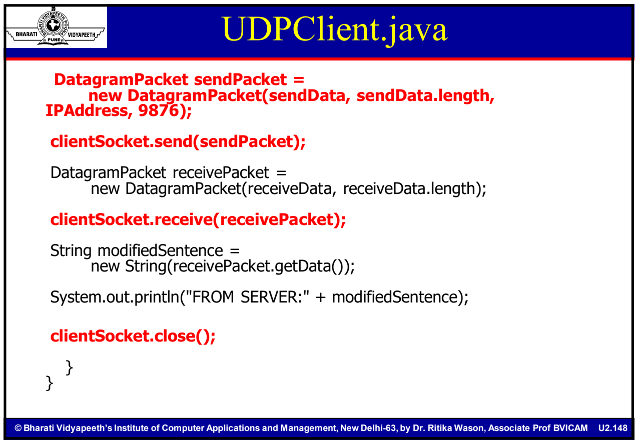
JAVA UDP Sockets

- In Package java.net
  - java.net.DatagramSocket
    - ✓A socket for sending and receiving datagram packets.
    - ✓Constructor and Methods
      - DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
      - void receive(DatagramPacket p)
      - void send(DatagramPacket p)
      - void close()



UDPClient.java

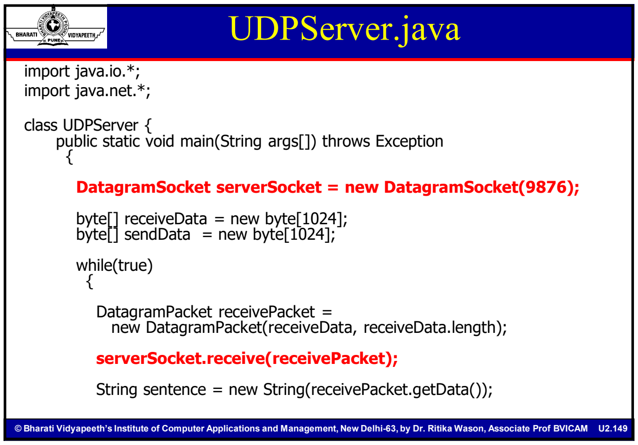
```
import java.io.*;
import java.net.*;

  class UDPClient {
     public static void main(String args[]) throws Exception
     {

       Scanner inFromUser =
         new Scanner(System.in);

       DatagramSocket clientSocket = new DatagramSocket();

       InetAddress IPAddress = InetAddress.getByName("hostname");

       byte[] sendData = new byte[1024];
       byte[] receiveData = new byte[1024];

       String sentence = inFromUser.nextLine();

       sendData = sentence.getBytes();
```



UDPClient.java

```
  DatagramPacket sendPacket =
       new DatagramPacket(sendData, sendData.length,
IPAddress, 9876);

 clientSocket.send(sendPacket);

 DatagramPacket receivePacket =
       new DatagramPacket(receiveData, receiveData.length);

 clientSocket.receive(receivePacket);

 String modifiedSentence =
       new String(receivePacket.getData());

 System.out.println("FROM SERVER:" + modifiedSentence);

 clientSocket.close();

   }
}
```

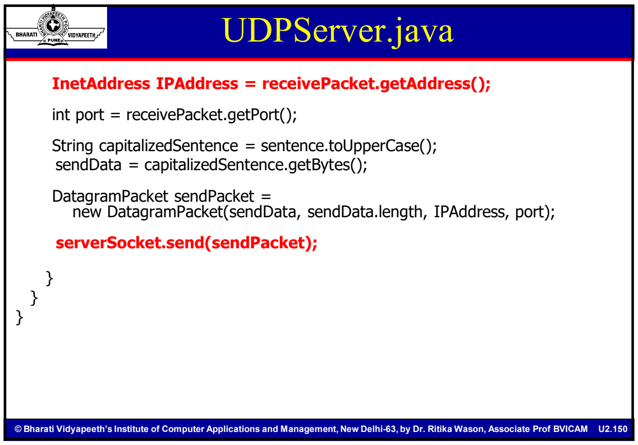
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.148



UDPServer.java

```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception
      {

        DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData  = new byte[1024];

        while(true)
          {

            DatagramPacket receivePacket =
               new DatagramPacket(receiveData, receiveData.length);

            serverSocket.receive(receivePacket);

            String sentence = new String(receivePacket.getData());
```

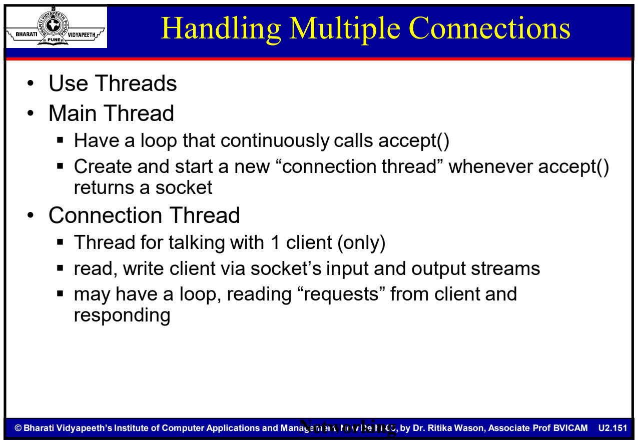
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.149



UDPServer.java

```
     InetAddress IPAddress = receivePacket.getAddress();

     int port = receivePacket.getPort();

     String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();

     DatagramPacket sendPacket =
        new DatagramPacket(sendData, sendData.length, IPAddress, port);

      serverSocket.send(sendPacket);

    }
  }
}
```

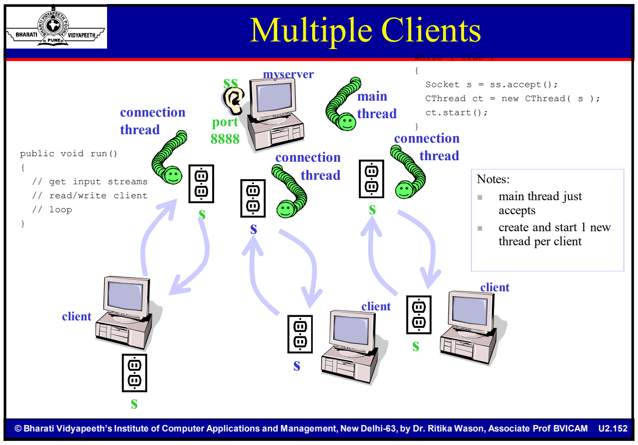
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Associate Prof BVICAM U2.150



Handling Multiple Connections

- Use Threads
- Main Thread
  - Have a loop that continuously calls accept()
  - Create and start a new “connection thread” whenever accept() returns a socket
- Connection Thread
  - Thread for talking with 1 client (only)
  - read, write client via socket’s input and output streams
  - may have a loop, reading “requests” from client and responding

Networking



Multiple Clients

```
while ( true )
{
  Socket s = ss.accept();
  CThread ct = new CThread( s );
  ct.start();
}
```

```
public void run()
{
  // get input streams
  // read/write client
  // loop
}
```

Notes:

- main thread just accepts
- create and start 1 new thread per client