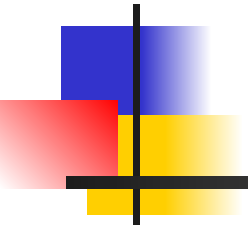
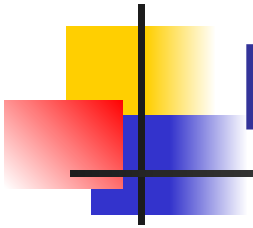


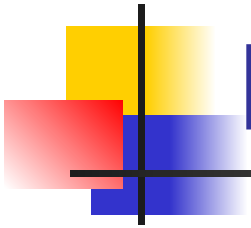
Session

Introduction to Threads



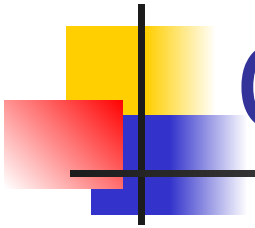
Review

- Whenever an error is encountered while executing a program, an `Exception` occurs.
- An `Exception` occurs at run time in a code sequence.
- Every exception that is thrown must be caught, or the application terminates abruptly.
- Exception handling allows combining error processing in one place.
- Java uses the `try` and `catch` block to handle exceptions.



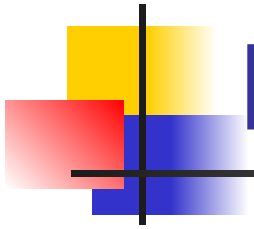
Review Contd…

- The statements in the `try` block throw exceptions and the `catch` block handles them.
- Multiple `catch` blocks can be used together to process various exception types separately.
- The `throws` keyword is used to list the exception that a method can throw.
- The `throw` keyword is used to indicate that exception has occurred.
- The statements in the `finally` block are executed irrespective of whether an exception occurs or not.



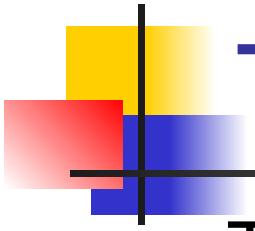
Objectives

- *Define a thread*
- *Define multithreading*
- *List benefits of multithreading*
- *Create threads*
- *Discuss thread states*
- *Manage threads*
- *Explain how to set thread priorities*
- *Describe a daemon thread*



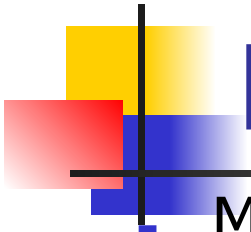
Multitasking Vs Multithreading

- `Multitasking` is the ability to run one or more programs concurrently.
- Operating system controls the way in which these programs run by scheduling them.
- Time elapsed between switching of programs is minuscule.
- `Multithreading` is the ability to execute different parts of a program called `threads`, simultaneously.



Thread

- Thread is the smallest unit of executable code that performs a particular task.
- An application can be divided into multiple tasks and each task can be assigned to a thread.
- Many threads executing simultaneously is termed as Multithreading.
- Appears that the processes are running concurrently, but it is not so.

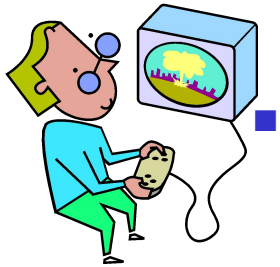


Benefits of Multithreading

- Multithreading requires less overhead than multitasking.
  - In multitasking, processes run in their own different address space.
  - Tasks involved in multithreading can share the same address space.
- Inter-process calling involves more overhead than inter-thread communication.
- Multithreading allows us to write efficient programs that make maximum use of CPU.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

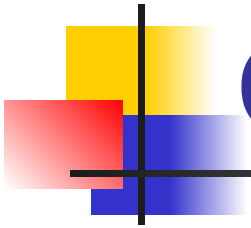
Applications of thread

- Playing sound and displaying images simultaneously



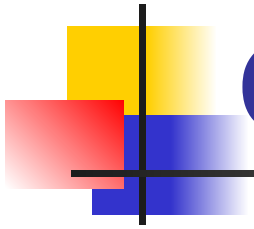
- Displaying multiple images on the screen

- Displaying scrolling text patterns or images on the screen



Creating threads

- When Java programs execute, there is always one thread running and that is the main thread.
  - It is this thread from which child threads are created.
  - Program is terminated when main thread stops execution.
  - Main thread can be controlled through `Thread` objects.
  - Reference of the main thread can be obtained by calling the `currentThread()` method of the `Thread` class.



Creating threads Contd…

- Thread objects can be created in two ways:
  - Declare a class that is a sub-class of the class `Thread` defined in `java.lang` package
    - `class` mythread `extends Thread`
  - Declare a class that implements the `Runnable` interface
    - `class` mythread `implements Runnable`
- While using applets, `Thread` class cannot be extended. Therefore one has to implement the `Runnable` interface.

Creating threads Contd…

- After a new thread has been initiated, we use the `start()` method to start the thread otherwise it is an empty `Thread` object with no system resources allocated.

```
Mythread t = new Mythread();
t.start();
```

- When `start()` method is invoked, the system resources required to run the thread is created and schedules the thread to run.
- It then calls the thread's `run()` method.

Creating threads Contd…

Example 1 –

Creating a thread by extending the `Thread` class

```
class MyThread extends Thread
{
    public static void main(String args[])
    {
        My        x = new MyThread();
        O
        S
    }

    public
    {
        Th
        O
  }
        }
}
```

Output

```
C:\WINDOWS\system32\cmd.exe

E:\Java\JavaExamples>javac MyThread.java

E:\Java\JavaExamples>java MyThread
This is the main thread
This is the child thread
This is the child thread
This is the child thread
This is the child thread
This is the child thread
```

Creating threads Contd…

Example2 –

Creating a thread by implementing the `Runnable` interface

```
class MyThread2 implements Runnable
{
    public static void main(String args[])
    public void run()
    {
        while
        {
th
thread");
        }
    }
}
```

Output

```
C:\WINDOWS\system32\cmd.exe

E:\Java\JavaExamples>javac MyThread2.java

E:\Java\JavaExamples>java MyThread2
This is the main thread
This is the child thread
This is the child thread
This is the child thread
This is the child thread
This is the child thread
```

Thread states

- **Born:** A newly created thread is in a born state.

- **Ready:** After a thread is created, it is in its ready state waiting for `start()` method to be called.

Thread states Contd…

- **Running:** Thread enters the running state when it starts executing

- **Sleeping:** Execution of a thread can be halted temporarily by using `sleep()` method. The thread becomes ready after sleep time expires.

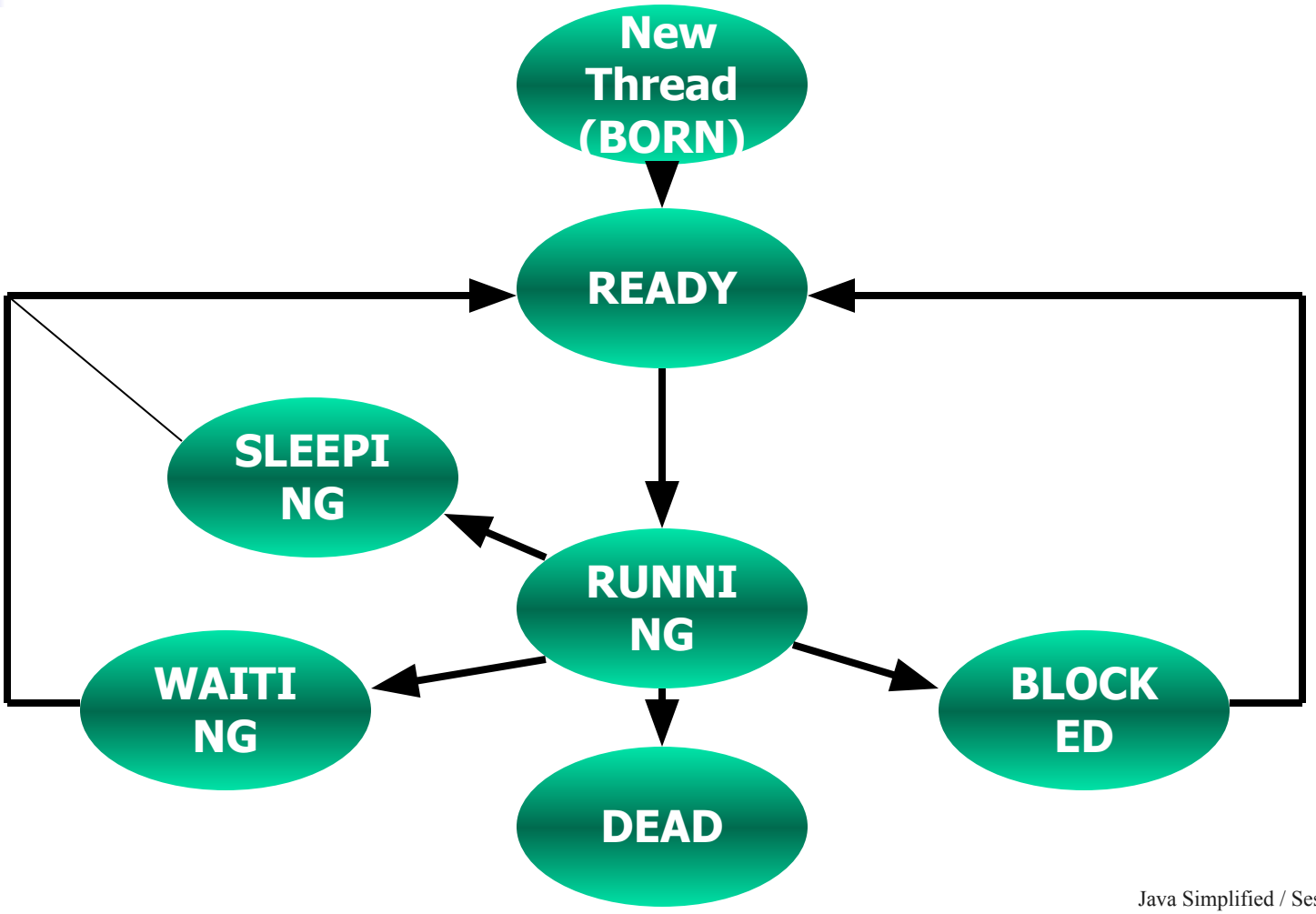
Thread states Contd…

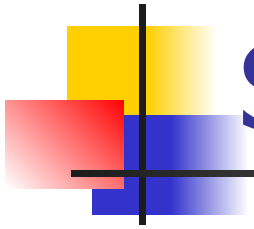
- **Waiting:** Thread is in waiting state if `wait()` method has been invoked. Used when two or more threads run concurrently.

- **Blocked:** The thread enters a blocked state when it waits for an event such as Input/Output operations.

- **Dead:** The thread enters the dead state after the `run()` method has finished or the threads `stop()` method is called.

Different thread states





Some methods of thread class

- `final boolean isAlive()`: returns true if the thread is alive.
- `final String getName()`: returns the name of the thread.
- `void start()`: used to start a thread by calling the method `run()`.

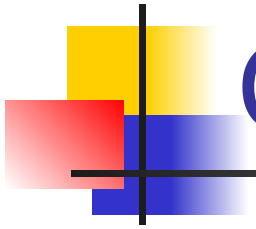
Some methods of thread class Contd…

- `final void join() throws InterruptedException`: waits for the thread to die.

- `static void yield()`: causes the currently executing thread to temporarily pause and allow other threads to execute.

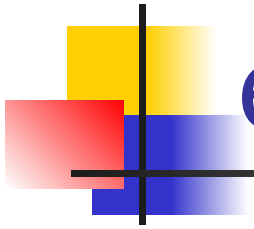
- `final void setName(String name)`: sets the name of the thread to the name that is passed as an argument.

Some methods of thread class Contd…

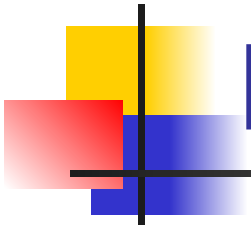


- `final boolean isDaemon()`: checks if the thread is a Daemon thread.
- `static int activeCount()`: returns the number of active threads.
- `static void sleep()`: used to suspend a thread for a certain period of time.

Conditions that prevent thread execution



- If thread is:
  - Not of highest priority
  - Put to sleep using `sleep()` method
  - Is waiting because `wait()` method was called
  - Explicitly yielded using `yield()` method
  - Blocked for file I/O

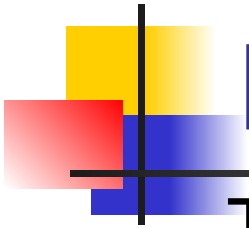


Managing threads

- Priorities for carrying out activities changes at times
  - eg :Planned to visit museum in the afternoon but due to toothache, had to go to doctor
- Similarly while programming, we may have to run a thread of higher importance without stopping or suspending the current running thread
- Thread priorities play an important role in such a situation.

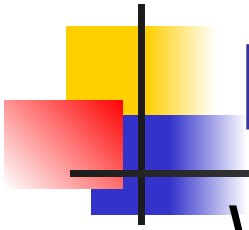
Managing threads Contd…

- Thread priorities in Java are constants defined in the `Thread` class.
  - `NORM_PRIORITY` – value is 5
  - `MAX_PRIORITY` – value is 10
  - `MIN_PRIORITY` – value is 1
- The default priority is `NORM_PRIORITY`
- Two methods used to change priority:
  - `final void setPriority(int newp)`: changes the thread's current priority.
  - `final int getPriority()`: returns the thread's priority.



Daemon threads

- Two types of threads in Java:
  - User threads: created by the user
  - Daemon threads: threads that work in the background providing service to other threads
    - e.g. – the garbage collector thread
- When user thread exits, JVM checks to find out if any other thread is running.
- If there are, it will schedule the next thread.
- If the only executing threads are daemon threads, it exits.



Daemon threads Contd…

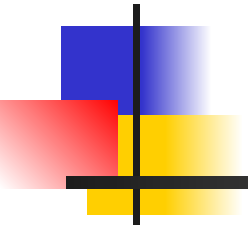
- We can set a thread to be a Daemon if we do not want the main program to wait until a thread ends.
- Thread class has two methods to deal with Daemon threads:
  - `public final void setDaemon(boolean value)` : sets a thread to be a daemon thread
  - `public final boolean isDaemon()` : checks if the given thread is a daemon thread

Daemon threads Contd…

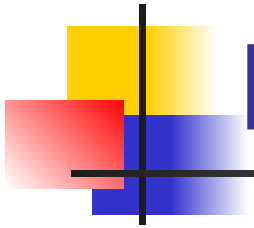
```
class TestDaemon implements Runnable
{
        Thread Objth1,Objth2;
        public TestDaemon()
        {
                Objth1 = new Thread(this);
                Objth1.start(
Objth2 = new Thread(th
Objth2.setDaemon(true
        }
        public void run()
        {
                System.out
unt());
                System.out
n());
System.out.println(Objth2.isDaemon());
        }
        public static void main(String args[])
        {
                new TestDaemon();
        }
}
```

Output

An example

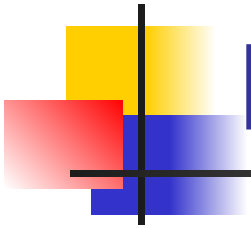


Multithreading and Garbage Collection



Review

- Multithreading allows programmers to write efficient programs that make the maximum use of the CPU.
- Java provides built-in support for multithreading in the form of classes and interfaces.
- When Java programs are executed, there is already one thread that is running and it is the main thread. This main thread is important for two reasons:
  - It is the thread from which child threads will be created.
  - Program is terminated when the main thread stops execution.
- Thread objects can be created in two ways:
  - Declare the class to be a sub-class of the `Thread` class where we need to override the `run()` method of the `Thread` class.
  - Declare a class that implements the `Runnable` interface. Then define the `run()` method.
- Each thread in a Java program is assigned a priority, and the Java Virtual Machine never changes the priority of a thread.

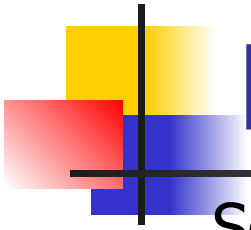


Review Contd…

- The default priority of a thread that is created is 5.
- Two of the constructors in the `Thread` class are:
  - `public Thread(String threadname)`
  - `public Thread( )`
- There are two types of threads in a Java program: User threads and Daemon threads.
  - The threads created by the user are called user threads.
  - The threads that are intended to be "background" threads, providing service to other threads are referred to as daemon threads.
- The `Thread` class has two methods that deal with daemon threads.
  - `public final void setDaemon(boolean on)`
  - `public final boolean isDaemon( )`

Objectives

- *Use multithreading with applets*
- *Use isAlive() and join()*
- *Explain the need for synchronization*
- *Discuss how to apply the keyword synchronized*
- *Explain the role of the methods wait(), notify() and notifyAll()*
- *Describe deadlocks*
- *Describe garbage collection*



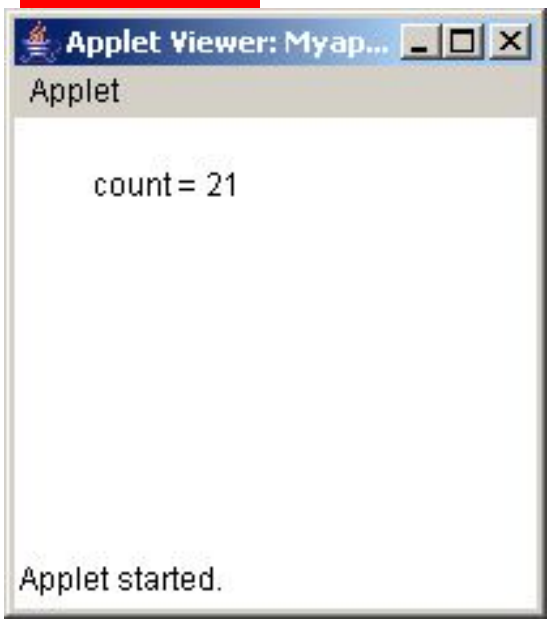
Multithreading with applets

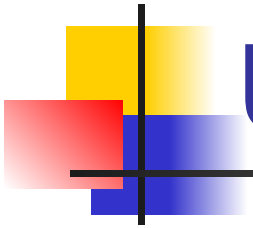
- Some instances of using multithreading on the web are:
  - Displaying scrolling marquees as banners
  - Displaying clocks or timers as part of web pages
  - Multimedia games
  - Animated images
- When an applet-based Java program uses more than one thread, it is called multithreading with applets.
- Since Java does not support multiple inheritance, it is not possible to subclass the `Thread` class directly in Applets.

Example

```
/*
<                                                    00>
<
*
i
i
p                                                    Runnable
{
    public void run()
    {
        for(count = 1; count <= 20; count++)
        {
            try
            {
                repaint();
                Thread.sleep(500);
            }
            catch (InterruptedException e)
            {}
        }
    }
    public void paint(Graphics g)
    {
        g.drawString("count = "+count,
30, 30);
    }
}
```

Output





Using isAlive() and join()

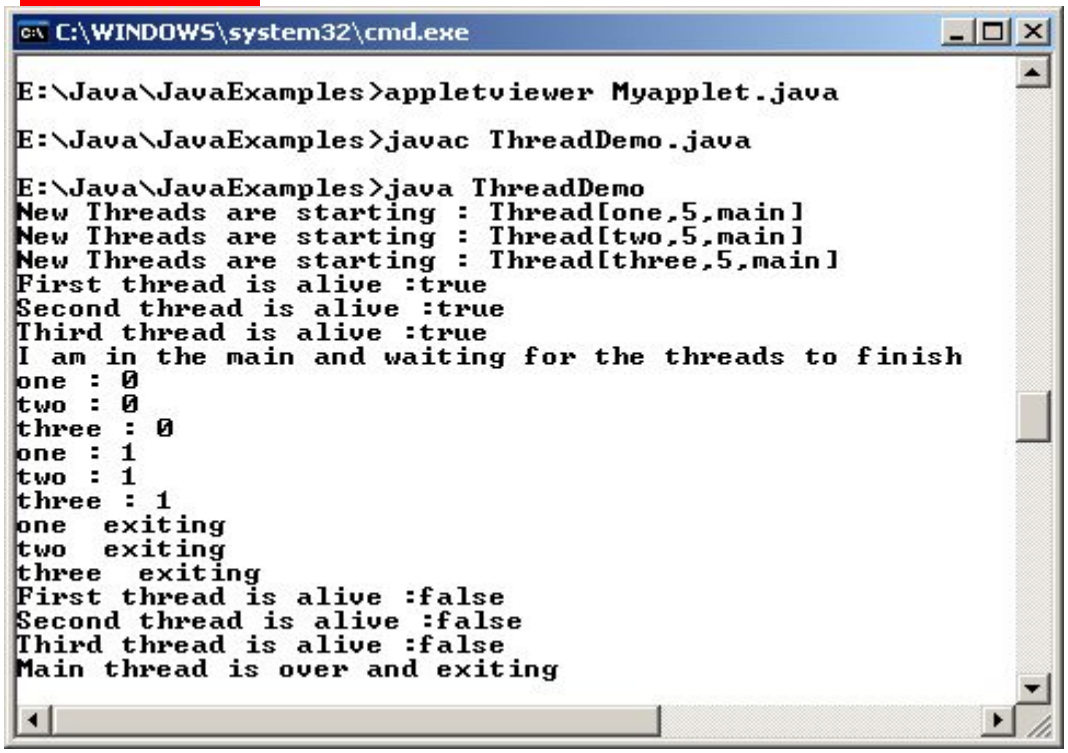
- The main thread should be the last thread to finish.
- We put the main thread to sleep for a long time within `main()` method and ensure that all the child thread terminate before main thread.
- There are two ways to find out if a thread has terminated. They are:
  - `isAlive()`
  - `join()`

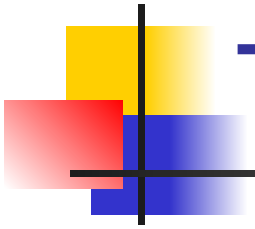
Example

```
class ThreadDemo implements Runnable
{
 public static void main(String [] args)
        {
                ThreadDemo Objnew1 = new ThreadDemo("one");
                ThreadDemo Objnew2 = new ThreadDemo ("two");
                ThreadDemo Objnew3 = new ThreadDemo ("three");
                System.out.println("First thread is alive :" + Objnew1.objTh.isAlive());
                System.out.println("Second thread is alive :" + Objnew2.objTh.isAlive());
                System.out.println("Third thread is alive :" + Objnew3.objTh.isAlive());
c               try
                {
                        System.out.println("I am in the main and waiting for the threads to
finish");
                        Objnew1.objTh.join();
                        Objnew2.objTh.join();
                        Objnew3.objTh.join();
                }
                catch(InterruptedException e)
                {
"                       System.out.println("Main thread is interrupted");
                }
                System.out.println("First thread is alive :" + Objnew1.objTh.isAlive());
                System.out.println("Second thread is alive :" + Objnew2.objTh.isAlive());
                System.out.println("Third thread is alive :" + Objnew3.objTh.isAlive());
                System.out.println("Main thread is over and exiting");
        }
}
        System.out.println(name + "  exiting");
```

Example Contd…

Output





Thread Synchronization

- At times, two or more threads may try to access a resource at the same time.
  - For example, one thread might try to read data from a file while another one tries to change the data in the same file
- In such a case, data may become inconsistent.
- To ensure that a shared resource is used by only one thread at any point of time, we use `synchronization`.

Thread Synchronization Contd…

- Synchronization is based on the concept of monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can enter a monitor.
- When one thread enters the monitor, it means that the thread has acquired a lock and all other threads must wait till that thread exits the monitor.
- For a thread to enter the monitor of an object, the programmer must invoke a method created using the `synchronized` keyword.
- Owner of the method has to exit from the method to give up the control.

Example

```
class One
        public void run()
        {
                objOne.display(number);
        }
}
class SynchMethod
{
        public static void main(String args[])
        {
                One objOne = new One();
                int digit = 10;
                Two objSynch1 = new Two(objOne,digit++);
                Two objSynch2 = new Two(objOne,digit++);
                Two objSynch3 = new Two(objOne,digit++);

                //wait for threads to end

                try
                {
                        objSynch1.objTh.join();
                        objSynch2.objTh.join();
                        objSynch3.objTh.join();
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted");
                }
  }
```

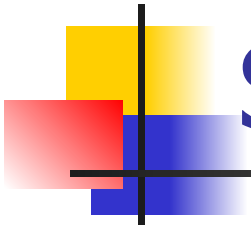
Example Contd..

Output

Race condition

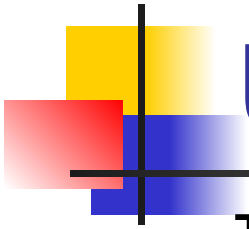
- If `synchronized` keyword is omitted from the previous example, all the threads can simultaneously invoke the same method, on the same object.
- This condition is known as **race condition.**
- Race conditions in a program are possible when
  - Two or more threads share data
  - They are reading and writing the shared data simultaneously

Output Without Synchronization



Synchronized Block

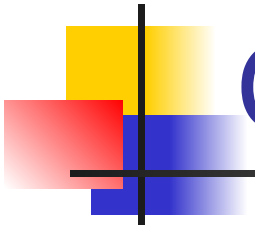
- It is not always possible to achieve synchronization by creating `synchronized` methods within classes.
- We can put all calls to the methods defined by this class inside a `synchronized` block.
- A `synchronized` block ensures that a method can be invoked only after the current thread has successfully entered object's monitor.
- The example shown earlier can be modified with the `synchronized` keyword used in the method `run()` of the class 'One'.



Using ‘wait-notify’ mechanism

- Java provides well designed inter-process communication mechanism using the `wait()`, `notify()` and `notifyAll()` methods.
- The methods are implemented as `final` methods in the class `Object`.
- `wait()`, `notify()` and `notifyAll()` can be called only from within a `synchronized` method.

Using 'wait-notify' mechanism Contd…



- `wait()` method tells the calling thread to exit and enter the `sleep` state till some other thread enters the monitor and calls the `notify()` method.
- `notify()` method wakes up the first thread that called `wait()`.
- `notifyAll()` wakes up or notifies all the threads that called `wait()`.
- Once all the thread are out of sleep mode, the thread that has the highest priority will run first.

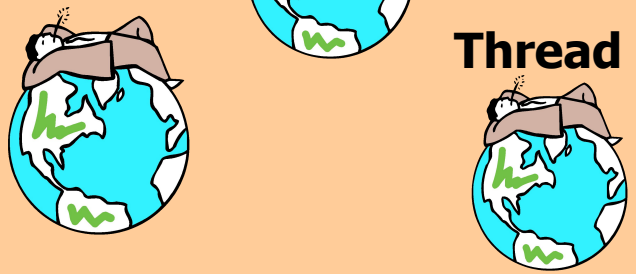
Using 'wait-notify' mechanism Contd…

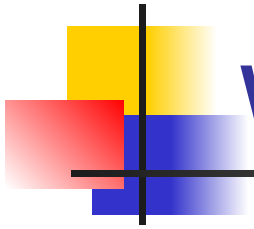
`notify()`

`notify()` **wakes up or notifies the first thread.**

`notifyAll()`

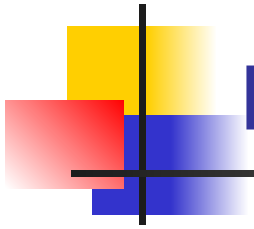
`notifyAll()` **wakes up or notifies all the threads that called wait( ) on the same object.**





wait()

- Points to remember while using the `wait()` method:
  - The calling thread gives up the CPU.
  - The calling thread gives up the lock.
  - The calling thread goes into the waiting pool of the monitor.



notify()

- Main points to remember about `notify()`:
  - One thread moves out of the waiting pool of the monitor and into the ready state.
  - The thread that was notified must reacquire the monitor's lock before it can proceed since it was in sleep state and no longer had the control of the monitor.

Example

```
class Philo
{
        Cho
        int

        Phi
        {




        }
        pub
        {


eating");
        }

         pu
        {


thinking");
        }
```

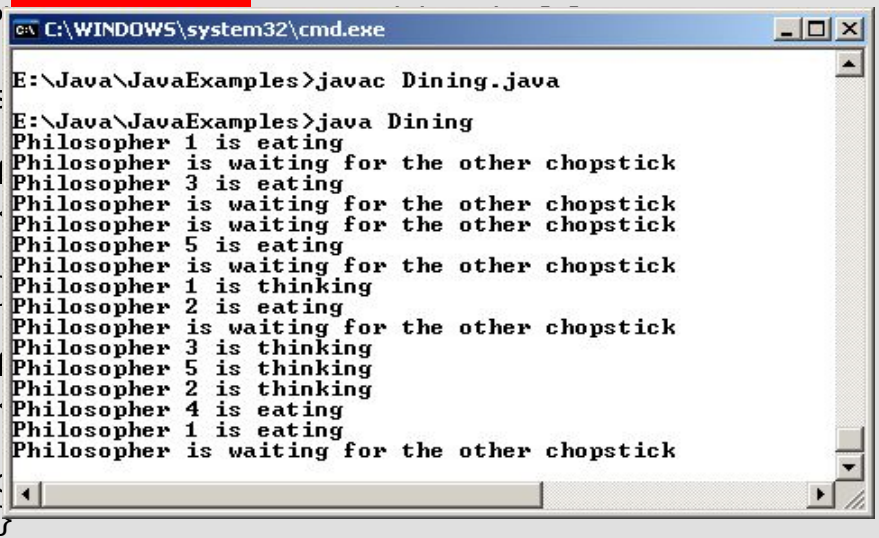
```
class ChopStick
{
        boolean available;
        ChopStick()
        {
                available = true;
        }
        public synchronized void takeup()
        {
                while(!available)
                {
                        try
                        {
                                System.out.println("Philosopher is waiting for the
other chopstick");
                                wait();
                        }
                        catch(InterruptedException e)
                        { }
                }
                available = false;
        }
        public synchronized void putdown()
        {
                available = true;
                notify();
        }
}
```

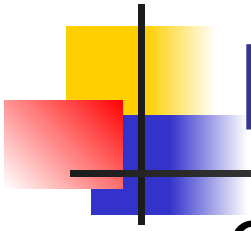
Example Contd…

```
public void run()
{
        while(true)
class Dining
{
        static Ch                    icks = new ChopStick[5];
        static Ph
        public s
        {
                f
                f
Philosopher(co
                }

                for (int count = 0;count <= 4;count++)
                        philos[count].start( );
        }
}
}
//end of class
```

Output





Deadlocks

- Occurs when two threads have a circular dependency on a pair of synchronized objects.
  - For example: one thread enters the monitor on object ‘ObjA’ and another thread enters the monitor on object ‘ObjB’.
  - If the thread in ‘ObjA’ attempts to call any synchronized method on ‘ObjB’, a deadlock occurs.

Example

```
public class DeadlockDemo implements Runnable
{
        public static void main(String args[])
        {
DeadlockDemo grabIt;

Dea

        public synchronized void run()
Dea     {
                try                                                     1);
                {                                                       2);
                        Thread.sleep(500);
                }
                catch(InterruptedException e)
                 {
                        System.out.println("error
occurred");
                }
                grabIt.syncIt();
        }
                        obj1.join();
        }
        catch(InterruptedException e)
        {
                System.out.println("error occurred");
        }
            System.exit(0);
        }
```

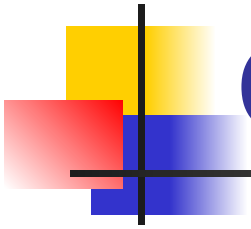
Example Contd…

```
public synchronized void syncIt()
        {
                try
                {
                        Thread.sleep(500);
                System.out.println("Sync");
                }
                catch(InterruptedException e)
                {
                System.out.println("error occurred");
                }
                System.out.println("In the syncIt()
r
}
```

Output

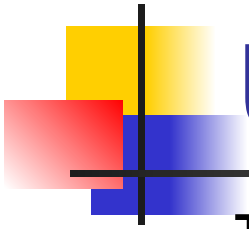
Garbage collection

- It is a process whereby the memory allocated to objects, which are no longer in use, may be reclaimed or freed.
- Java automatically frees the memory that is no longer required.
- Thus programmers do not have to worry about garbage collection at all.
- An object becomes eligible for garbage collection if there are no references to it or if it has been assigned to null.



Garbage Collection Contd…

- Garbage collector runs as a separate low priority thread.
- Garbage collector can be invoked by invoking that instance's `gc ()` method.
- There is no guarantee that garbage collection will take place right then.



Using the finalize method

- Java provides a way that is similar to C++ destructors, which can be used for cleaning up process before the control returns to the operating system.
- The `finalize()` method if present will be executed prior to garbage collection only once per object. Syntax of the method is:
  - `protected void finalize() throws Throwable`
- Referrences cannot be garbage collected; only objects are.

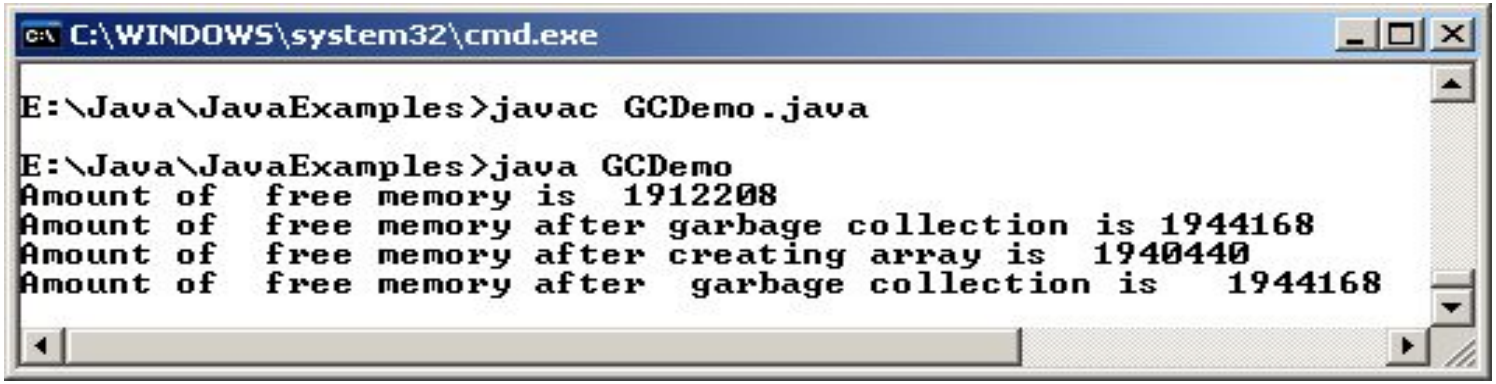
Example

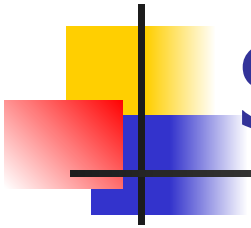
```
class GCDemo
{
        public static void main(String args[])
        {
                int count;
                long num;
                Runtime objRun = Runtime.getRuntime();
                Long values[] = new Long[200];
                 System.out.println("Amount of  free memory is  " + objRun.freeMemory());
                objRun.gc();
                System.out.println("Amount of  free memory after garbage collection is "+ objRun.freeMemory());
                for(num = 10000,count = 0; count < 200; num++,count++)
                {
                        values[count] = new Long(num);
                }
                System.out.println("Amount of  free memory after creating array is  "+ objRun.freeMemory());

                for (count = 0;count < 200 ; count++)
                {
                        values[count] = null;
                }
                objRun.gc();
                System.out.println("Amount of  free memory after  garbage collection is   "+ objRun.freeMemory(
));
        }
}
```

Example Contd…

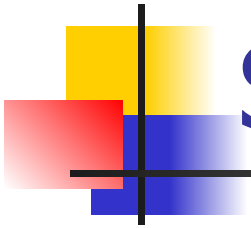
Output





Summary

- Data may get corrupted when two or more threads access the same variable or object at the same time.
- The method `isAlive()` returns true if the thread upon which it is called is still running.
- The method `join()` will wait until the thread on which it is called terminates.
- `Synchronization` is a process that ensures that the resource will be used by only one thread at a time.
- `Synchronization` does not provide any benefit for single threaded programs. In addition, their performance is three to four times slower than their non-synchronized counterparts.
- The method `wait()` tells the calling thread to give up the monitor and enter the sleep state till some other thread enters the same monitor and calls the method `notify()`.



Summary Contd…

- The method `notify()` wakes up or notifies the first thread that called `wait()` on the same object.
- The method `notifyAll()` wakes up or notifies all the threads that called `wait()` on the same object.
- A deadlock occurs when two threads have a circular dependency on a pair of `synchronized` objects.
- Garbage collection in Java is a process whereby the memory allocated to objects, which are no longer in use, may be reclaimed or freed.
- The garbage collector runs as a low priority thread and we can never predict when it will collect the objects.