Threads

What is a thread?

- Sun defines a thread as a single sequential flow of control within a program.
- A threads is a sequence of lines of code that execute within a process.
- It is sometimes referred to as an execution context or a lightweight process.
- Thread based multitasking environments allow a single program to perform two or more tasks simultaneously.
- The thread in java is a realization of OS level thread. In other words, the thread in java is created using native methods which in turn create threads based on the OS.

Test your knowledge

- Can you distinguish between a Thread and a Process at OS level?

Uses of threads in application

- In client-server based systems, the server program creates threads that allows it to respond to multiple users at the same time.
- GUI programs have a separate thread to gather user’s interface events from the host OS.
- Do other things while waiting for slow I/O operations.
- Animations

`java.lang.Thread` class

- So far programs that we created used just one thread.
- Java SE provides 2 classes for working with thread called **`Thread`** and **`ThreadGroup`**.
- When a Java Virtual Machine starts up, that is when an application starts to execute, there is usually a single thread called **`main`**.
- The main threads continues to execute until
  - **`exit()`** method is called
  - all threads (non-daemon threads) have died
- A deamon thread is a special type of thread that runs in background. When JVM exits, only remaining thread which will be running are daemon threads. By default all the threads that are created in java are non-daemon threads. *More on this later*

Printing main thread

```
public class ThreadTest{
public static void main(String s[])
{
System.out.println(“Hello”);
System.out.println(
      Thread.currentThread().getName());

}
}
```

Prints : main

`currentThread()` is a _______ method of _________class.
`getName()` is ________________ method of ___________ class

`Thread` class constructors

- There are two ways to create threads
  - By extending **`Thread`** class
    - Constructors that will be called in such case could be
      - **`Thread()`**
      - **`Thread(String name)`**
  - By creating **`Thread object`** and passing **`Runnable`** instance.
    - Constructors that will be called in such case could be
      - **`Thread(Runnable target)`**
      - **`Thread(Runnable target, String name)`**

Creating Threads by extending

```
class SimpleThread extends Thread {
public void run(){
/* code that is executed when
thread executes */
}}
```

Creating the `SimpleThread` object

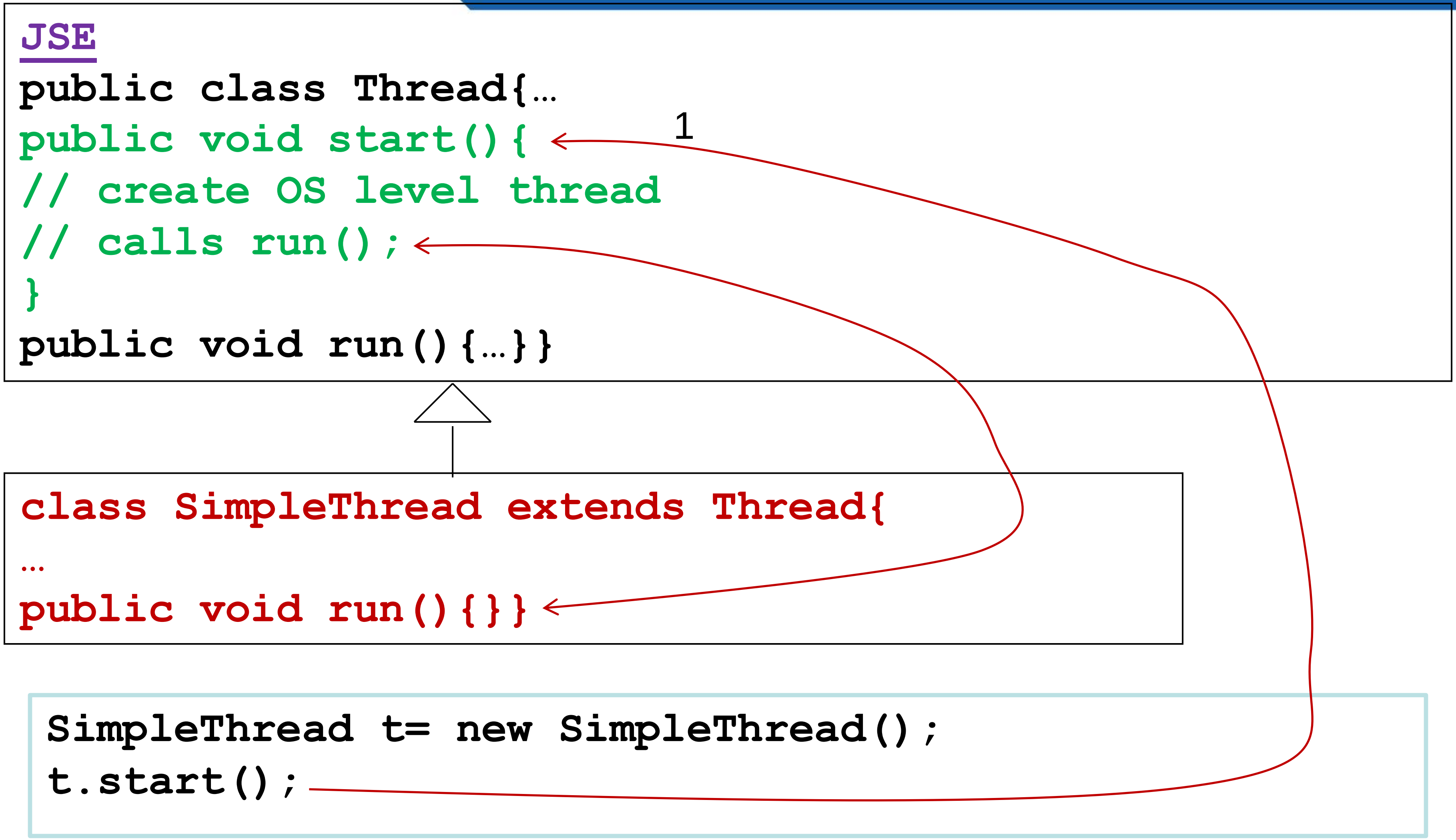
```
SimpleThread t= new SimpleThread();
t.start(); // Calls run() method of SimpleThread
```

- We override the `run()` method and put the code that needs to be executed when the thread runs, in the `run()` method.
- To call run method we call `start()` !

Starting a thread

- Since **`SimpleThread`** inherits from the **`Thread`** class, **`start()`** method of the **`Thread`** class gets inherited into the **`SimpleThread`** class.
- The **`start()`** method of the **`Thread`** class creates OS level thread.
- After this it calls the **`run()`** method.
- Since the **`run()`** method is overriden , **`run()`** method of the **`SimpleThread`** is called.

Starting a thread



JSE

```
public class Thread{…
public void start(){
// create OS level thread
// calls run();
}
public void run(){…}}
```

```
class SimpleThread extends Thread{
…
public void run(){}}
```

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

Example : create Thread through inheritance

- The code below tests if the given number is prime or non-prime, even or odd.
- Thread no. 1 : user created thread that tests for prime or non-prime.
- Thread no. 2 : main thread that tests for even or odd.

```
import java.util.*;
class PrimeOddThread extends Thread {
int num;
public static void main(String str[]){
      PrimeOddThread c= new PrimeOddThread ();
      Scanner scan= new Scanner(System.in);
      int i=scan.nextInt();
      c.num=i;
      c.start();
      if(c.odd())
     System.out.println("Odd");
      else  System.out.println("Even");
}
public void run() {
      if( prime())
   System.out.println("Prime");
   else      System.out.println("Non-Prime");
 }
public boolean prime(){
      for(int i=2;i<num/2;i++){
      if(num%i==0)            return false;}
      return true;}
public boolean odd(){
      if(num%2==0) return false;
      else return true;
 }}
```

- The code is likely to print : Even first and then Prime (if prime and even number like 2 is entered)
- But strictly speaking, one can never guarantee whether Even will be printed or Prime first. It is totally up to the OS thread scheduler to choose which thread to execute first.

Test your understanding

- What will happen if you call **`run()`** method instead of **`start()`** method in the previous example?

Problem with the first way – Using `Runnable`

- The way the thread was created (in the previous example) thread requires your class to inherit from **`Thread`** class.
- This means that this class cannot inherit from any other class.
- Another way to program threads is by implementing **`Runnable`** interface.
- **`Runnable`** interface has one method
  - **`public void run()`**
- Second way to create thread is by
  1. Creating a class that implements **`Runnable`** interface- that is overriding the **`run()`**
  2. Creating a Thread class instance and passing **`Runnable`** instance. *Recall the 2nd set of constructors of Thread*

Understanding working with `Runnable`

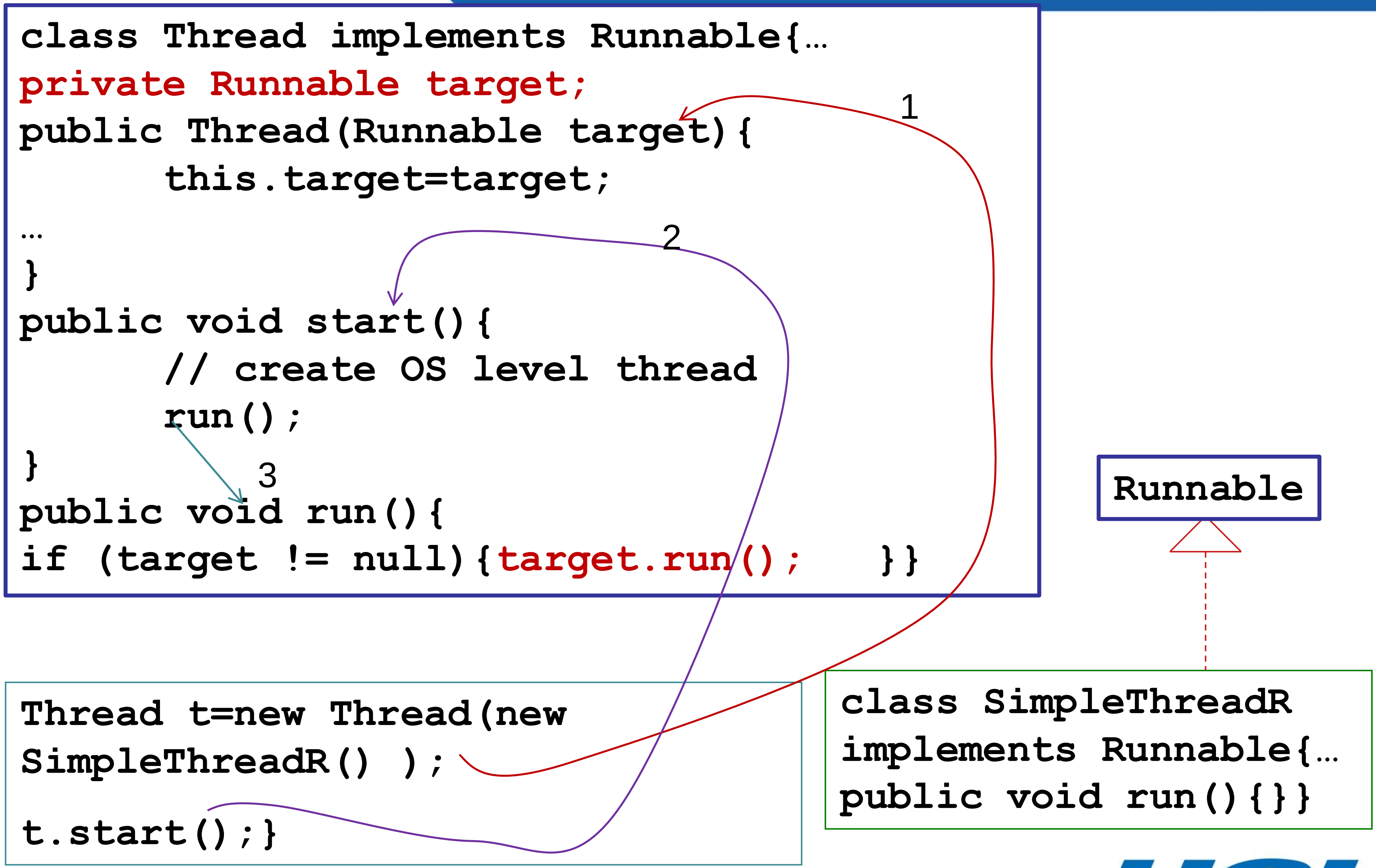
```
class SimpleThreadR implements Runnable{
public void run(){…}
}
```

//Creation of thread – using a constructor that expects a Runnable object

```
Thread t= new Thread(new SimpleThreadR() );
t.start(); // calls the run method of SimpleThreadR
```

- `Thread` class has a member called `target` which is of type `Runnable`.
- When the `Thread` constructor as above is invoked, the `target` member is assigned to the instance that is passed via constructor.
- When `start()` is called on the thread instance OS level thread gets created and then `run()` of Thread is called.
- `run()` of `Thread` in turn calls `run()` of target member.

Callback working behind the scene



```
class Thread implements Runnable{…
private Runnable target;
public Thread(Runnable target){
      this.target=target;
…
}
public void start(){
      // create OS level thread
      run();
}
public void run(){
if (target != null){target.run();   }}
```

```
Thread t=new Thread(new
SimpleThreadR() );

t.start();}
```

```
class SimpleThreadR
implements Runnable{…
public void run(){}}
```

Code snippet of `Thread` class

```
    /
    public synchronized void start() {
        if (started)
            throw new IllegalThreadStateException();
        started = true;
        group.add(this);
        start0();
    }

    private native void start0();

    /**
     * If this thread was constructed using a separate
     * <code>Runnable</code> run object, then that
     * <code>Runnable</code> object's <code>run</code> method is
called;
     * otherwise, this method does nothing and returns.
     * <p>
     * Subclasses of <code>Thread</code> should override this method.
     *
     * @see     java.lang.Thread#start()
     * @see     java.lang.Thread#stop()
     * @see     java.lang.Thread#Thread(java.lang.ThreadGroup,
     *          java.lang.Runnable, java.lang.String)
     * @see     java.lang.Runnable#run()
     */
    public void run() {
      if (target != null) {
          target.run();
      }
    }
```

Example :create Thread using Runnable

```
import java.util.*;
class PrimeOddThread implements Runnable {
int num;

public static void main(String str[]){
   PrimeOddThread x=new PrimeOddThread();
   Thread c= new Thread (x );
   Scanner scan= new Scanner(System.in);
   int i=scan.nextInt();
   x.num=i;
   c.start();
   if(x.odd())
        System.out.println("Odd");
   else
   System.out.println("Even");
}
public void run() {
    if( prime())
      System.out.println("Prime");
      else System.out.println("Non-Prime");
}

public boolean prime(){
   for(int i=2;i<num/2;i++){
   if(num%i==0)return false;}
   return true;}
   public boolean odd(){
   if(num%2==0) return false;
   else return true;
}
}
```

Naming Threads

- Every thread is given a name. If you don’t specify a name, a default name will be created.
- For example the main thread is named ‘main’.
- The default name of a user defined thread is ‘Thread-0’ for the first thread created, ‘Thread-1’ for the second and so on.
- To explicitly name the thread
  - Use constructor
    - **`Thread(Runnable target, String name) or`**
    - **`Thread(String name)`**
  - Or instance method
    - **`final void setName(String name)`**

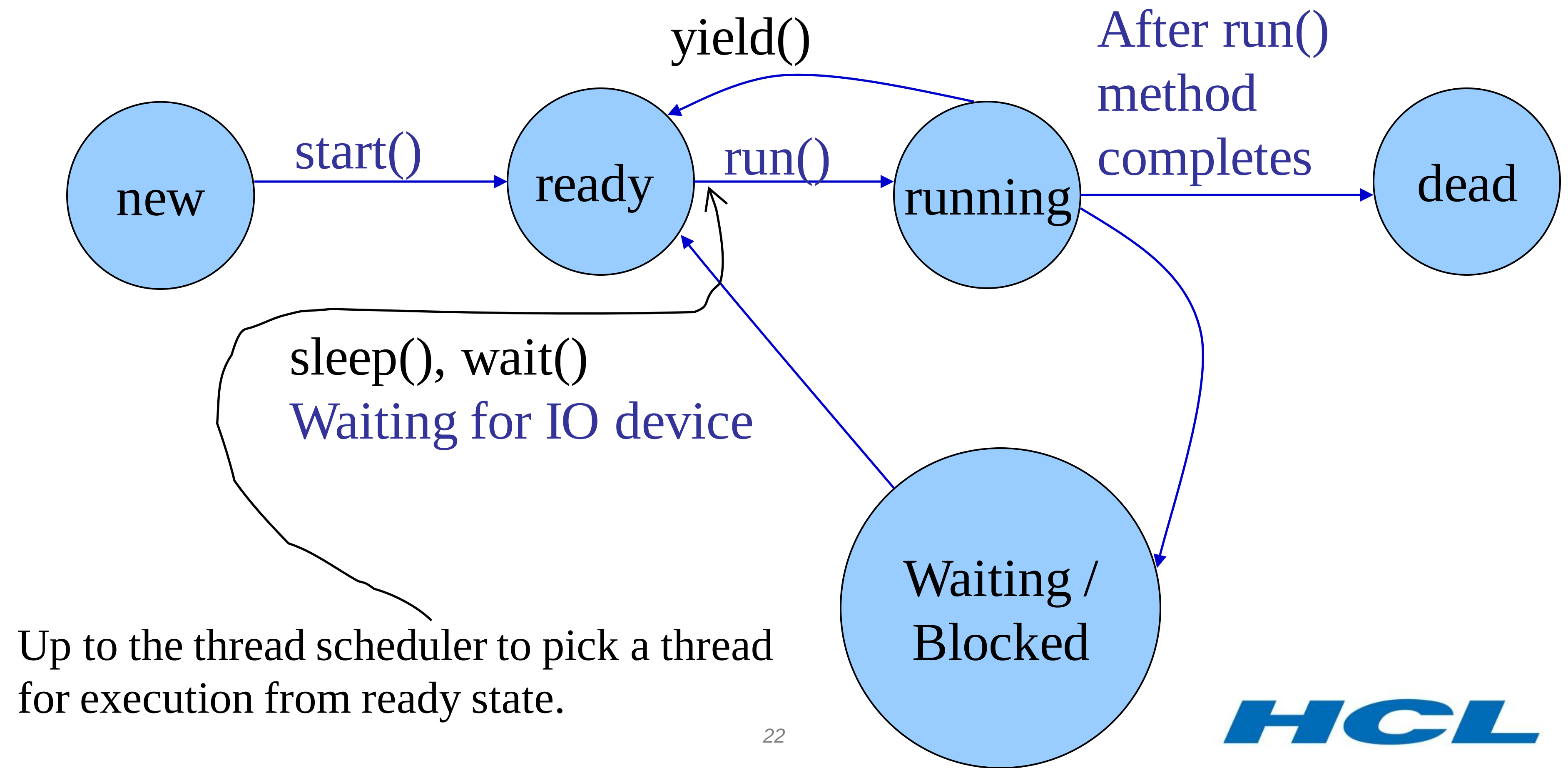
What is the method used to get the name? What is the method used to get the current thread?

Lifecycle of a thread

- Thread life cycle begins when a new thread instance is created.
- Then start() method on thread is called. Calling the start() method does not mean that the thread runs immediately. There may be many threads waiting to run. It is up to Thread scheduler of the OS to decide which thread will run first. We can't control the scheduler from the java program! So when we have several threads (of same priority), we cannot determine which thread will run when.
- Also, just because a series of threads are started in a particular order doesn't mean they'll run in that order.
- When start() method is called a new thread of execution starts (with a new call stack).The thread moves from the new state to the ready state.
- When the thread gets a chance to execute, its run() method will be called.
- A thread may be *blocked* - waiting for a resource (like printer etc). Once the thread gets the resource, it again moves to ready state.

Lifecycle

- When a thread completes its run() method, the thread is considered dead.
- `final boolean isAlive()` can be used to check if the thread is alive or dead.
- An attempt to start a dead thread throws `IllegalThreadStateException`



yield()
After run() method completes
new
start()
ready
run()
running
dead
sleep(), wait()
Waiting for IO device
Waiting / Blocked
Up to the thread scheduler to pick a thread for execution from ready state.

Putting a thread to sleep

- `sleep()` causes the currently executing thread to sleep (temporarily cease execution) for the minimum of specified number of milliseconds.
- **`static void sleep(long millis) throws InterruptedException`**
- **`static void sleep(long millis, int nanos) throws InterruptedException`**
- Note that `sleep()` is `static` method.
- **`InterruptedException`** is a checked exception. A sleeping thread can be interrupted so that it's sleep can be broken. This can be done using `interrupt()`

Coming up!

Example for sleep

```
public class Appear implements Runnable{
 char c[]={‘H',‘E',‘L',‘L‘,’O’};
 public void run() {
      int i=0;
      try{
      while (i<5){
      System.out.print(c[i++]);
      /* will sleep for at least a second */
      Thread.sleep(1000);
 }
 }catch(InterruptedException e){}
 }
 public static void main(String str[]){
      Thread t =new Thread(new Appear());
      t.start();  }}
```

Can you guess what this code does?

Interruption

- A thread can be interrupted while it is sleeping or waiting. This can be done by calling **`interrupt()`** on it. In other words, when the thread is calling **`sleep(), join() or wait()`** methods, it can be interrupted .
- When **`interrupt()`** is called, an **`InterruptedException`** exception will be thrown. . So the catch handler block will be executed. **`sleep(), join() or wait()`** methods throw **`InterruptedException`**
- The **`interrupt`** status will be reset before the **`InterruptedException`** exception is thrown.
- Methods that can be used to test if the current thread has been interrupted
  - **`static boolean interrupted() : interrupted status`** of the thread is cleared by this method
  - **`boolean isInterrupted(): interrupted status`** of the thread is unaffected

join() and wait() coming up!

Example : `interrupt`

The code prints "Have a nice day" every 1 sec until user presses an a key. When user presses a key, an **`InterruptedException`** is thrown and catch handler is called. The **`break`** in catch handler makes the thread come out of while loop and hence **`run()`** method returns

```
import java.util.*;
public class Doll extends Thread {
   public void run() {
      while(true){
      try {
            System.out.println("Have a nice day");
            Thread.sleep(1000);
      } catch(InterruptedException e) { break; }      }}
public static void main(String args[]) {
     Doll d = new Doll();
     d.start();
     Scanner scan = new Scanner(System.in);
     String s=scan.nextLine();
      if(s!=null)   d.interrupt();
} }
```

`join()`

- When a thread calls `join( )` on another thread instance, the caller thread will wait till the called thread finishes execution.
- **`final void join() throws InterruptedException`**
- `join()` can also be specified with some timeout , in which case the thread waits at most milliseconds for this thread to die.
- If timeout specified as 0 means the thread will wait forever
- **`final void join(long millis) throws InterruptedException`**
- **`final void join(long millis, int nanos) throws InterruptedException`**
- A thread waiting because of `join()` can also be interrupted. Hence the method throws a `InterruptedException`

Example: `join()`

If we want to guarantee that Prime (or Non-Prime) is printed before Even(or Odd) then the thread printing Even(or Odd), that is main thread, can wait till the prime thread finishes the execution.

```
class Join implements Runnable{
int num;
public void run() {
 if( prime()) System.out.print("Prime");
 else         System.out.print("Non-Prime"); }

public boolean prime(){
   for(int i=2;i<num/2;i++){
   if(num%i==0)return false;
   }
   return true;   }

public boolean odd(){
   if(num%2==0)return false;
   else return true;    }
public static void main(String str[]){
    Join s=new Join();
    Thread t= new Thread(s);
    s.num=55;
    t.start();

    try{     t.join();} //main thread waits for t to
    finish

    catch(InterruptedException  e){}


    boolean b=s.odd();

    if(b)

    System.out.print(" and Odd");
    else     System.out.print("and Even");

}}
```

Thread priorities

- A thread is always associated with a priority based on the OS
- For simplicity Java assigns priority as a number between 1 and 10, inclusive of 10. 10 is the highest priority and 1 is the lowest.
- This is the only way to influence the scheduler’s decision as to the thread execution sequence to some extent.
- At any given time, the highest-priority thread will be chosen to run. However, this is not guaranteed. The thread scheduler may choose to run a lower-priority thread to avoid starvation.
- Threads that were created so far were created with a default priority of 5.
- The new threads inherit the priority from the thread that created it.

Methods for thread priorities

- Setting and getting priority
  - **`final void setPriority(int newPriority) throws IllegalArgumentException`**
  - **`final int getPriority()`**
  - Note that **`setPriority() throws IllegalArgumentException`** if number passed to the method is not between 1 and 10 (inclusive of 10).
  - Also the priority cannot be greater than maximum permitted priority of the thread's thread group (even if the **`newPriority`** specifies a number greater than the thread's thread group priority)
- **`static`** constants to set priorities:
  - **`Thread.MIN_PRIORITY (1)`**
  - **`Thread.NORM_PRIORITY (5)`** → default
  - **`Thread.MAX_PRIORITY (10)`**

Selfish threads

- In non-pre-emptive OS, if a thread that executes with high priority and has long execution time does not voluntarily leave the CPU to give a chance for other threads with similar priority to execute, such a thread is called a selfish thread.
- It is recommended that threads with high priority (be unselfish)
- They must either call **`Thread.sleep()`** or **`Thread. yield()`** method.
  - **`static void yield()`**
- **`yield()`** method causes the currently executing thread object to temporarily pause and allow other threads to execute.

Why threads synchronization?

1. Let us say we have an Account class which has withdraw and deposit methods.
2. For each transaction (deposit or withdraw) a thread is created.
3. Let us visualize what happens when two people simultaneously withdraw from the same account object .
4. Let us assume that there is Rs. 1000 in the account

Thread A calls withdraw(500)

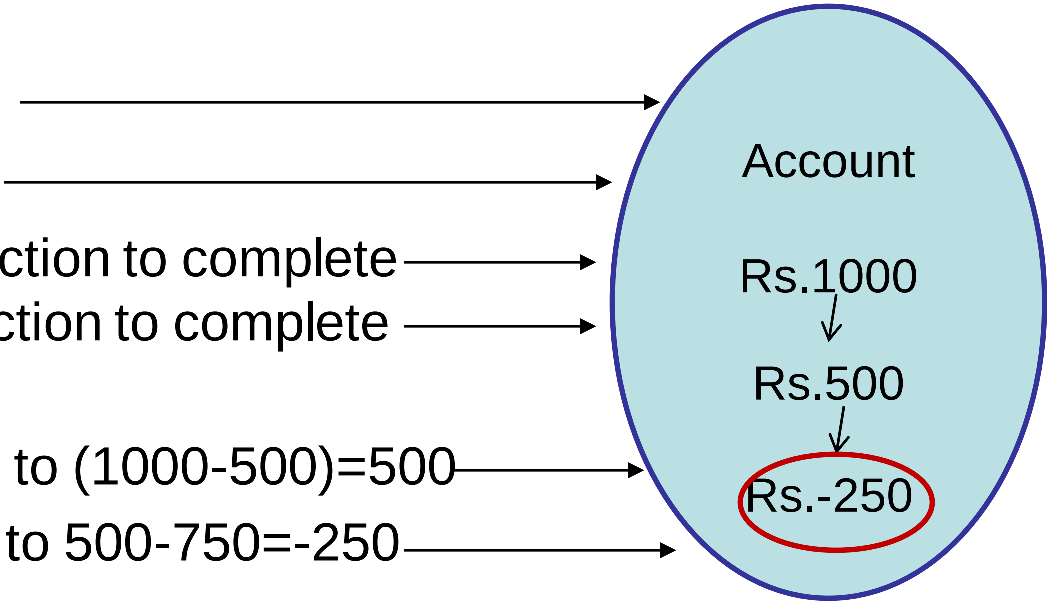
Thread B calls withdraw(750)

Thread A waits for the transaction to complete

Thread B waits for the transaction to complete

Thread A updates the money to (1000-500)=500

Thread B updates the money to 500-750=-250



Code demonstrating the same…

Code without synchronization

```
class Account{
   private int money;
   Account(int amt){
   //get amt from database
      money=amt;}
   void withdraw(int amt){
      if(amt<money){
      try{
         /*
          sleep() here is used to simulating time to
         connect to other systems and performing IO
         operation
         */
      Thread.sleep(1000);
       money=money-amt;
       }catch(Exception e){}
   System.out.println("Received "+ amt  +" by " +
   Thread.currentThread().getName());
      }
   else
   System.out.println("Sorry "+
   Thread.currentThread().getName()+ "Requested amt ("+
   amt +") is not available.");
   System.out.println("Balance "+ money);
}
}
public class ThreadTest implements Runnable
{
   Account a;
   int amt;
```

```
public static void main(String str[]){
  Account lb= new Account(1000);
  new ThreadTest(lb,"A",500);
  new ThreadTest(lb,"B",750);
}

public ThreadTest(Account a,String name,int amt){
  this.a=a;
  this.amt=amt;
  new Thread(this,name).start();
}

public void run(){ a.withdraw(amt);}
}
```

Result:

```
Received 500 by A
Balance 500
Received 750 by B
Balance -250
```

Terms

- If two threads access the same object and each calls a method that changes the state of that object then data corruption could result. This is called **race condition.**
- Monitor is a block of code guarded by a mutual-exclusion semaphore (*mutex or intrinsic lock* or *monitor lock*). A thread that wants to enter the monitor must have mutex.
- Only one thread can own the mutex at a time. Other threads which wants to enter the critical code must wait till the thread that acquired mutex releases it.
- In Java, every *object* has one and only one monitor and mutex associated with it.

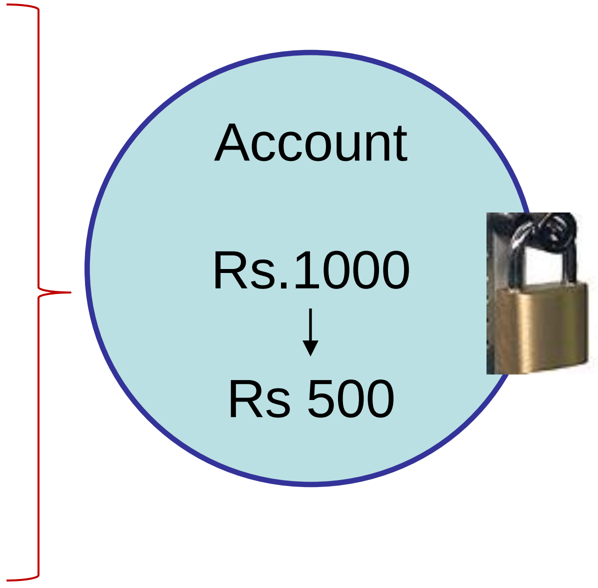
Solution to the Account problem

Thread A acquires mutex for the Account object and calls withdraw(500)

Thread B waits for A to release the mutex

Thread A waits for the transaction to complete

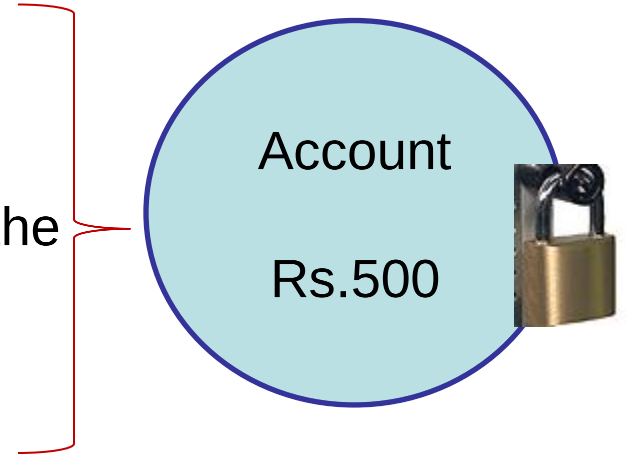
Thread A updates the money to (1000-500)=500 and releases the mutex



Thread B acquires mutex for the object and calls withdraw(750)

Thread B gets the message that it cannot withdraw the requested amount

Thread B releases the mutex



`synchronized`

- Java implements this concept by using `synchronized` keyword.
- `synchronized` can be used in 2 ways
  - A method can be marked `synchronized`
  - A block of code can be marked `synchronized`→ `synchronized` statements
- A thread enters a `synchronized` code only if no other thread is executing any other a `synchronized` code on that object. Therefore, two invocations of synchronized methods on the same object cannot interleave.
- Once a thread enters a `synchronized` code, it acquires a mutex. It is released only when the thread leaves the `synchronized` code.

Correcting code using `synchronized`

- If an object is visible to more than one thread, all reads or writes to that object's non final attributes should be done through synchronized methods.

- Approach 1: Add synchronized keyword to withdraw and other critical methods of the Account object.

```
synchronized void withdraw(int amt)
```

- Approach 2: Use `synchronized` statements by explicitly locking the object before calling critical methods of the Account object.

```
 public void run(){
   synchronized(a)    {
    a.withdraw(amt);  }
   }
```

Tell me how?

How does `static synchronized` method work?

- When they are invoked, since a `static` method is associated with a class, the thread acquires the monitor lock for the `Class` object associated with the class.
- Thus access to class's static fields is controlled by this lock. Note that a lock on static method has no effect on any instances of that class.

But what is `Class`?

- Every class loaded in Java has a corresponding instance of java.lang.Class representing that class. This object contains all static members of the class X
- This Class object is in the method area (that is logically part of the heap).
- The heap contains a pointer to the location of the object's class in the Method Area

Deadlock

- A deadlock occurs when two or more threads wait for each other to release the lock on a particular object.
- Scenario:

  A needs to transfer money from Account 1 to Account 2.

  B needs to transfer money from Account 2 to Account 1.

  - Thread A acquires intrinsic lock on Account 1
  - Thread B acquires intrinsic lock on Account 2
  - Thread A waits for intrinsic lock for Account 2 to be released
  - Thread B waits for intrinsic lock of Account 1 to be released
  - Ends up in a DEADLOCK

Code leading to deadlock

```
public class Lock{
  public static void main(String[] args) {
    final Account resource1 = new Account(2000);
    final Account resource2 = = new Account(2000);

 // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {

 public void run() {
        // Lock resource 1
        synchronized (resource1) {
      System.out.println("Thread 1:locked resource 1.
updating the balance");
      try {
            Thread.sleep(50);
          } catch (InterruptedException e) {    }
```

```
// Lock resource 2
synchronized (resource2) {
System.out.println("Thread 1: locked resource 2.
updating the balance "); }
 }
 } };

// t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
      // Lock resource 2
        synchronized (resource2) {
          System.out.println("Thread 2: locked
resource. updating the balance ");
try {
            Thread.sleep(50);
          } catch (InterruptedException e) {
          }
```

```
// Lock resource 1
synchronized (resource1) {
            System.out.println("Thread 2:
locked resource.updating the balance ");
          }
        }
}};

t1.start();
t2.start();
  }
}
```

If all goes as planned, deadlock will occur, and the program will never exit.

Some ways to prevent deadlock

- Avoid deadlock by first locking all the resources in some predefined sequence at the start itself before starting on any thing critical.

```
void method1(){
synchronized(resource1){
synchronized(resource2){
…}}
}
```

```
void method2(){
synchronized(resource1){
synchronized(resource2){
…}}
}
```

Sequencing locking

The predefined sequence must be carefully thought out.
Will this code always work?

```
public void transferMoney(Account fromAccount,
Account toAccount, double amt) {
    synchronized (fromAccount) {
      synchronized (toAccount) {
        if (fromAccount.bal>amt) {
           fromAccount.debit(amt);
           toAccount.credit(amt);
         }
}     }  }
```

What if a transfer of money happens from account 111 to account 222 and at the same time a transfer of money happens from account 222 to account 111? Do see the deadlock occurring?

One solution to this is by locking the accounts in the increasing order. That is first lock 111 and then 222.

Few more points

- Synchronized methods of super class can be overridden to be unsynchronized.
- Constructors cannot be declared as synchronized because only the thread that creates an object should have access to it while it is being constructed
- A non-static inner class can lock it's containing class using a synchronized block.
- Methods in the interface cannot be declared as **`synchronized`**.
- The locking does not prevent threads from accessing **`unsynchronized`** methods.
- Synchronized methods are also called **thread-safe** methods.

Daemon threads

- So far the threads that we have been creating are called foreground threads. A program continues to execute as long as it has at least one foreground (non-daemon) thread that is alive.
- The daemon threads are also called service threads. They are used for background processes that will continue only as long as the active threads of the program are alive.
- Daemon threads cease to execute when there are no non-daemon threads alive because when VM detects that the only remaining threads are daemon threads, it exits.
- The threads we have seen so far are non-daemon.
- Example: garbage collector thread is a daemon thread that runs with a low priority.
- **`final void setDaemon(boolean on)`** : must be called before the thread is started.
- **`boolean isDaemon()`**

Example for daemon threads

The code prints “"Have a nice day‟ for as long as main methods runs. The main method finishes execution after printing the last letter O.

```
public class Doll extends Thread {
   public void run()   {
      while(true)    {
      try {
            System.out.println("Have a nice day");
            Thread.sleep(1000);
      } catch(InterruptedException e) { break; }
          }
}
public static void main(String args[])    {
     Doll d = new Doll();
     d.setDaemon(true);
     d.start();
     char c[]={'H','E','L','L','O'};
int i=0;
     try{
     while (i<5){
     System.out.print(c[i++]+" ");
     Thread.sleep(1000);
     }
     }catch(InterruptedException e){}

  System.out.println("End of main method");
      }
 }
```

ThreadGroup

- **`ThreadGroups`** object have collection of threads.
- This helps in manipulating a group of threads all at a time instead of manipulating each of them individually . For instance all the threads in a collection can be started together.
- Every Java thread is a member of some thread group. When a thread is created, unless it is explicitly put in a thread group, the runtime system automatically places the new thread in the same group as the thread that created it.
- When a Java application first starts up, the Java runtime system creates a **`ThreadGroup`** named "**`main`**".

`ThreadGroup` Members

- Constructors
  - **`ThreadGroup(String name)`**
  - **`ThreadGroup(ThreadGroup parent, String name)`**
- **`final ThreadGroup getThreadGroup()`**

  Is a method of **`Thread`** class that returns the thread group the calling thread belongs to
- Methods to enumerate:
  - **`int enumerate(Thread[] list)`**
  - **`int enumerate(Thread[] list, boolean recurse)`**
- Methods with respect to the group
  - **`int activeCount()`**
  - **`ThreadGroup getParent()`**
  - **`String getName()`**
  - **`boolean parentOf(ThreadGroup g)`**

Thread constructors with ThreadGroup

- `Thread(ThreadGroup group, Runnable target)`
- `Thread(ThreadGroup group, Runnable target, String name)`
- `Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
- `Thread(ThreadGroup group, String name)`

Methods :threads as a group

- `int getMaxPriority()`
- `void setMaxPriority(int pri)`
- `void interrupt()`
- `void destroy()`
- `boolean isDaemon()`
- `void setDaemon(boolean daemon )`
- `void list()` (Prints information about this thread group to the standard output.)

```
class ThreadGroupDemo{
public static void main (String [] args){
   ThreadGroup tg = new ThreadGroup ("group 1");
   Thread t1 = new Thread (tg, "thread 1");
   Thread t2 = new Thread (tg, "thread 2");
   Thread t3 = new Thread (tg, "thread 3");

   tg = new ThreadGroup ("group 2");
   Thread t4 = new Thread (tg, "thread 4");
   tg = Thread.currentThread ().getThreadGroup ();
   int agc = tg.activeGroupCount ();
   System.out.println ("Active thread groups in " +
   tg.getName () +" thread group: " + agc);
   tg.list ();                                }}
```

```
Active thread groups in main thread group: 2
java.lang.ThreadGroup[name=main,maxpri=10]
Thread[main,5,main]
java.lang.ThreadGroup[name=group 1,maxpri=10]
java.lang.ThreadGroup[name=group 2,maxpri=10]
```

Inter-thread communication

- Inter-thread communication is required when execution of one thread depends on another thread's task.
- In such case, the second thread intimates or notifies the first thread when it has finished some task that the first thread is waiting for.
- The best suited situation to understand this is a producer-consumer problem.
- A producer thread produces something which consumer thread consumes. A producer and consumer thread can run independently.
- What we need to ensure however is that producer makes sure that it has produced enough for consumer to consume. If producer has not produced then consumer will have to wait till producer finishes.

`wait()`

- Consumer thread needs to check if the item to be consumed is there. Otherwise it has to wait. So a method is required to wait on the object where is item produced (object that is consumed is member of the object that is going to be consumed).
- Therefore some wait method has to be in the Object class.

```
final void wait() throws InterruptedException
final void wait(long timeout) throws
InterruptedException
public final void wait(long timeout, int nanos)
throws InterruptedException
```

- The first methods causes current thread to wait until either another thread invokes the **`notify()`** or the **`notifyAll()`** for this object,
- The second method waits for **`notify()`** or the **`notifyAll()`** for this object or waits for a specified amount of time which ever happens first.

notify()

- After the production of item, the Producer thread has to intimate the thread or threads which are waiting for the production. The production happens for a member of the an object.
- But the producer instead of directly notifying all the threads, just notifies the object. Since object has list of all waiting (consumer) threads with it (as a result of the wait calls), object awakens all the consumer threads.

`final void notify()`

→Wakes up a single thread that is waiting on this object's lock. The choice is arbitrary .

`final void notifyAll()`

→Wakes up all threads that are waiting on this object's lock.

Monitor

- Now one every important thing to bear in mind is that both producer and consumer thread have to own a monitor before producing or consuming.
- When **`wait()`** is called, the thread releases ownership of this monitor and waits until another thread notifies. Therefore the waiting threads has to wait until can re-obtain ownership of the monitor and resume execution
- When the thread that calls **`notify()`** the lock is not released.
- Therefore the producer thread after producing calls **`wait()`** to relinquishes the lock on the object to allow consumer to consume.
- Similarly the consumer thread must notify the producer that it has finished consuming by calling **`notify()`**. This awakens the producer thread which is on **`wait()`**.

Exceptions thrown by `wait` and `notify`

- Overloaded **`wait()`** methods throw a checked exception which is **`InterruptedException.`**
- *So what does it mean when a method throws* **`InterruptedException?`**
- Overloaded **`wait() and notify()`** methods also throw an uncheck exception which is **`IllegalMonitorStateException.`**

  (Please note that generally when method is defined only checked exceptions are listed in the exception list.)
- **`IllegalMonitorStateException`** is thrown if these methods are not called from a synchronized context.

Example: Producer/Consumer

- Example has one producer thread (tree) and one consumer threads (earthworm).
- Producer produces random amount apples if the apple count is less than 100 and notifies the consumer. It then waits until apple count is <100.
- Consumer consumes random amount of apples if apple count is equal to or above the count needed and then notifies the producer. If the required apples are not available it waits until producer notifies it.
- `(int)(Math.random()*100)+1` produces a double number between 0 and 100.

```
public class ProducerConsumer implements Runnable{
int  apples; // mainatins the count of the apples
public static void main(String s[]){
// Creating 3 threads
   ProducerConsumer pc = new ProducerConsumer();
   Thread producer= new Thread(pc, "Tree");
   Thread consumer1= new Thread(pc, “Worm");
```

```
// Starting 3 threads
   producer.start();
   consumer1.start();
   consumer2.start();
}

//Producer calls this method
synchronized void produce(){
   while(true){
   //produce apples only if there are <=100 apples
   if(apples>100)
   try{
   System.out.println("Waiting for apples to be eaten");
   wait();
   }
   catch(InterruptedException e){}
```

```
try{
   int i=(int)(Math.random()*100)+1;
   Thread.sleep(i*10); // time taken to produce apples
   apples=apples+i;
   System.out.println("Produced apples ="+ apples);
   }catch(InterruptedException e){}
   notifyAll();
}}
//Consumers calls this method
synchronized void consume(){
   while(true){
   int i=(int)(Math.random()*100)+1;
   // consume only if there are enough apples
   if(apples>0 && apples<i)
   try{
   System.out.println(Thread.currentThread().getName()
   +" waiting for "+ (i-apples) +" more apples");
   wait();}catch(InterruptedException e){}
```

```
try{
   System.out.println(Thread.currentThread().getName()+ "
   busy eating  "+ i +" apples");
   Thread.sleep(i); // time taken to consume apples
   apples=apples-i;}
   catch(InterruptedException e){}
   notify();}
}


public void run(){
   if(Thread.currentThread().getName().equals("Tree"))
   produce();
   else consume();

}}
```

Sample output

```
Produced apples =91
Produced apples =164
Waiting for apples to be eaten
Worm busy eating  42 apples
Worm busy eating  27 apples
Worm busy eating  73 apples
Worm busy eating  17 apples
Worm waiting for 54 more apples
Produced apples =88
```

Test your understanding

Will the code work if we had another consumer, say man?

`notify()` arbitrary selects a single thread which has called **`wait()`**.

Let us analyze the code for **`consume()`** method. If man is on **`wait()`** and earthworm calls **`notify()`** then what happens if man awakens and gets a change to execute.

Test your understanding

```
if(apples>0 && apples<i)
try{
   int j=(i-apples);
   System.out.println(Thread.currentThread().getName()
   + " waiting for "+j+" more apples");
   wait();}catch(InterruptedException e){}}
try{
```

Man wakes up but does not check if apples are there before eating

```
   System.out.println(Thread.currentThread().getName()+
   " busy eating  "+ i +" apples");
   Thread.sleep(i);
   apples=apples-i;}
   catch(InterruptedException e){}
   notify();}}
```

Multiple producer and consumers

Similar to the problem in the previous case, even in case of multiple producers we have similar problem.
Cases where there could be multiple consumer and producer, while loops are used instead of if for verifying the condition.

Corrected code to allow multiple producer and consumer:

```
synchronized void produce(){
   while(true){
   ~~if(apples>100)~~
   while(apples>100){
   try{
   System.out.println("Waiting for apples to be eaten");
   wait();}catch(InterruptedException e){}}
   try{
   int i=(int)(Math.random()*100)+1;
   Thread.sleep(i*10);
   apples=apples+i;
   System.out.println("Produced apples ="+ apples);
   }catch(InterruptedException e){}
   notifyAll();}}
```

```
synchronized void consume(){
   while(true){
    int i=(int)(Math.random()*100)+1;
   ~~if(apples>0 && apples<i)~~
    while (apples>0 && apples<i){
   try{
   int j=(i-apples);
   System.out.println(Thread.currentThread().getName() +
   " waiting for "+j+" more apples");
   wait();}catch(InterruptedException e){}
}
try{
   System.out.println(Thread.currentThread().getName()+
   " busy eating  "+ i +" apples");
   Thread.sleep(i);
   apples=apples-i;}
   catch(InterruptedException e){}
   notify();}}
```