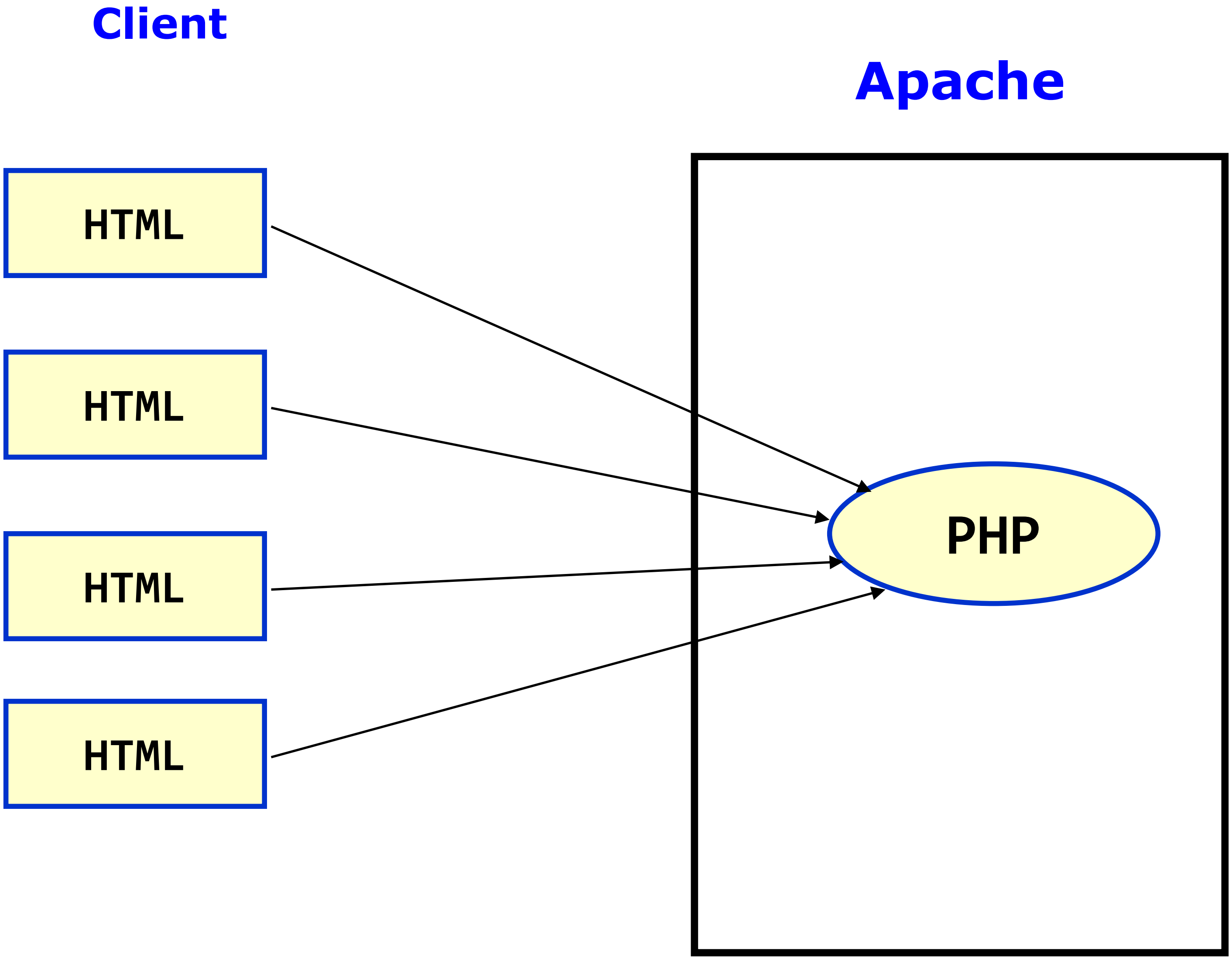
Multitasking

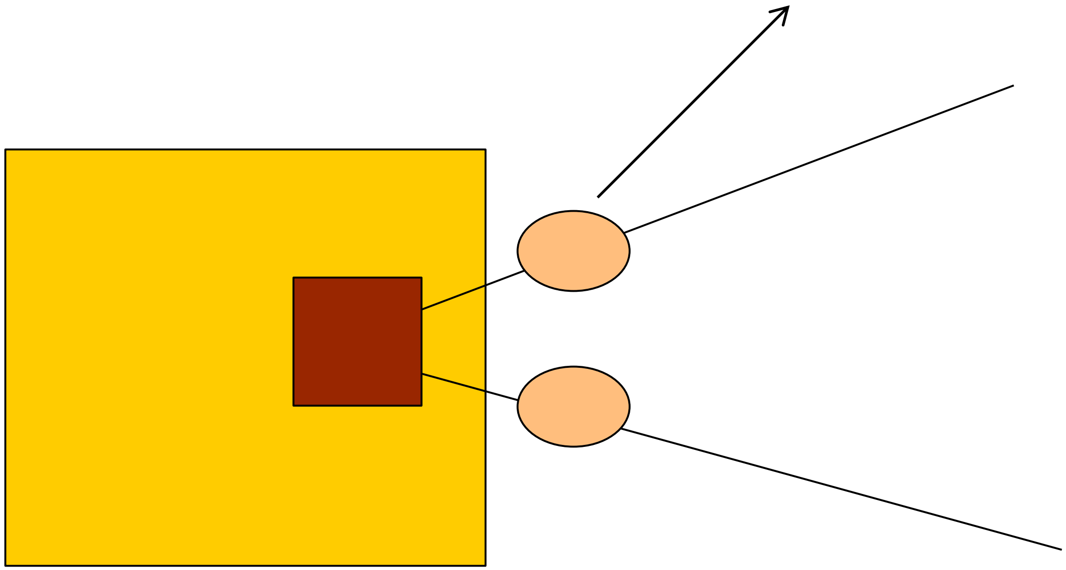
Multitasking

- Process Based
- Thread Based

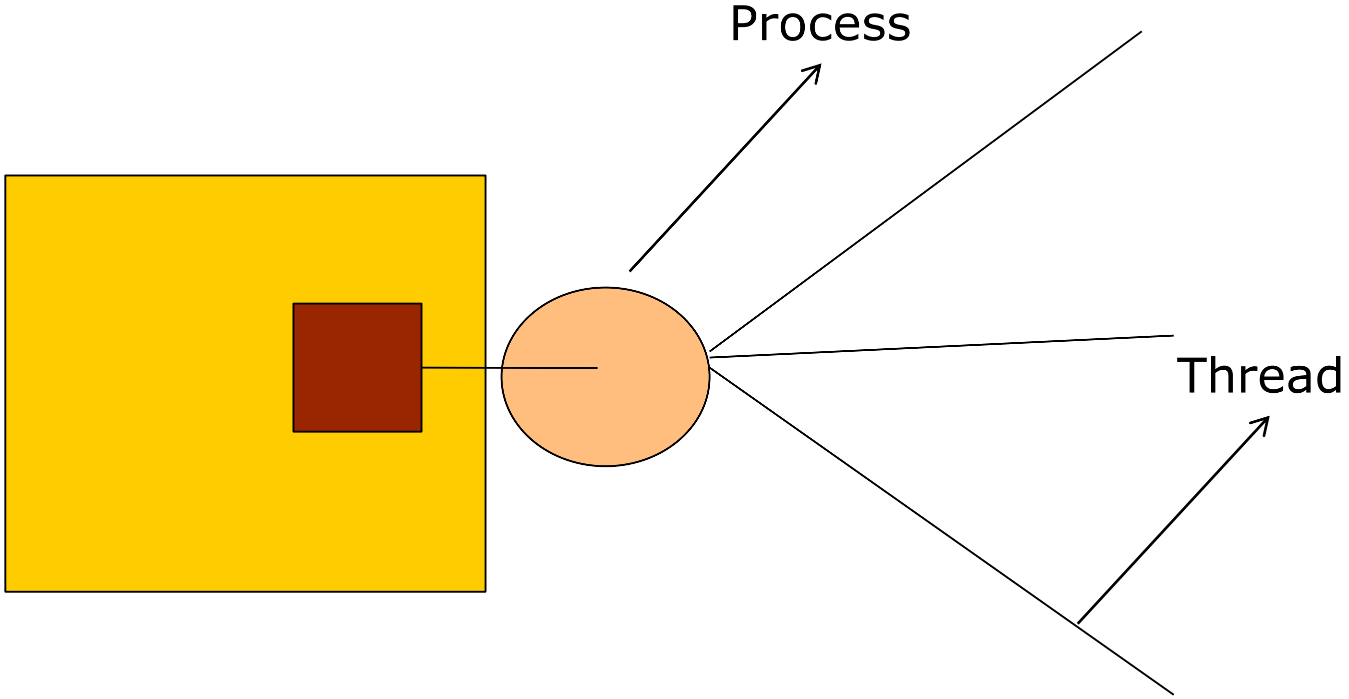
Web Server



Process Based Multitasking



Thread Based Multitasking

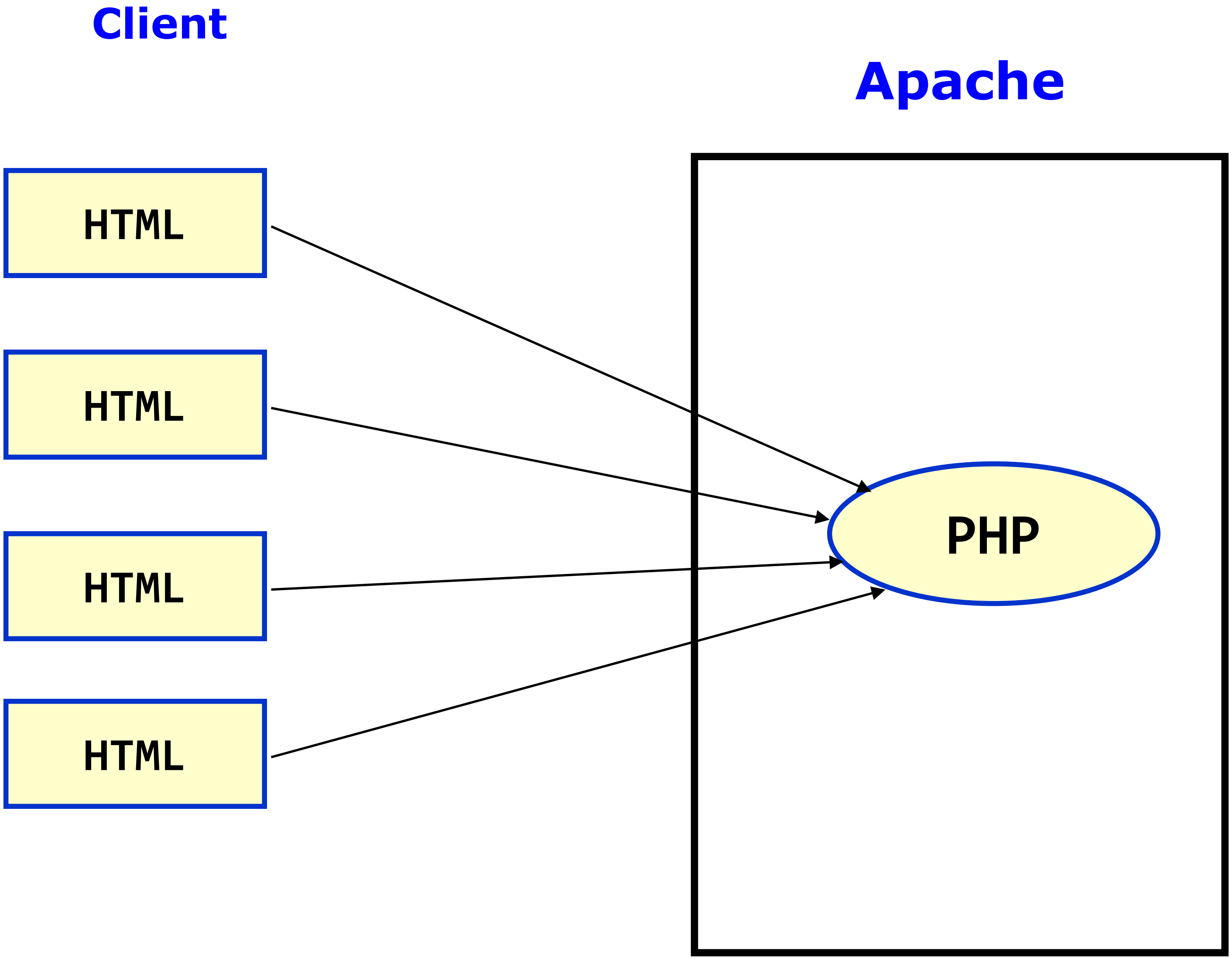


Multithreading

Why do we need threads?

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

Web Server



Client

Apache

HTML

HTML

HTML

HTML

PHP

Ways To Create a Thread

java.lang.*;

1) By implementing Runnable interface
2) By extending Thread class

Runnable inteface

- public void run()

Thread class

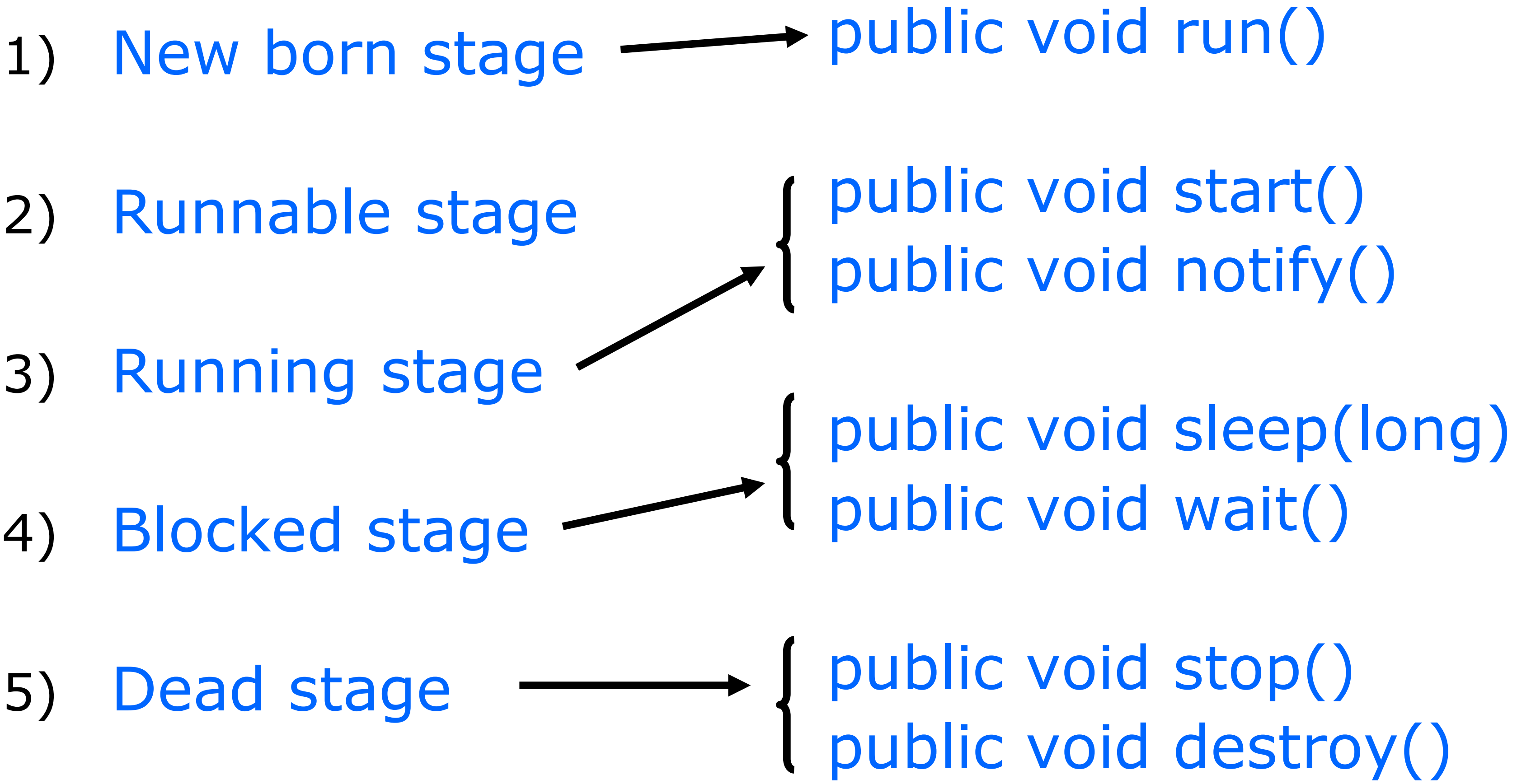
- public void run()

- public void start()
- public void notify()

- public void sleep(long)
- public void wait()

- public void stop()
- public void destroy()

Threads Life Cycle



1) New born stage → public void run()

2) Runnable stage

3) Running stage → public void start()
public void notify()

4) Blocked stage → public void sleep(long)
public void wait()

5) Dead stage → public void stop()
public void destroy()

Thread class

```
class  A
{
}
class B extends A
{

}
```

```
A   ob1  =  new A();

A   ob2  =  new B();
```

```
interface  A
{
}
class B implements A
{

}
```

```
A    ob1 = new A();    X

A    ob2 = new B();    ✓
```

Constructors

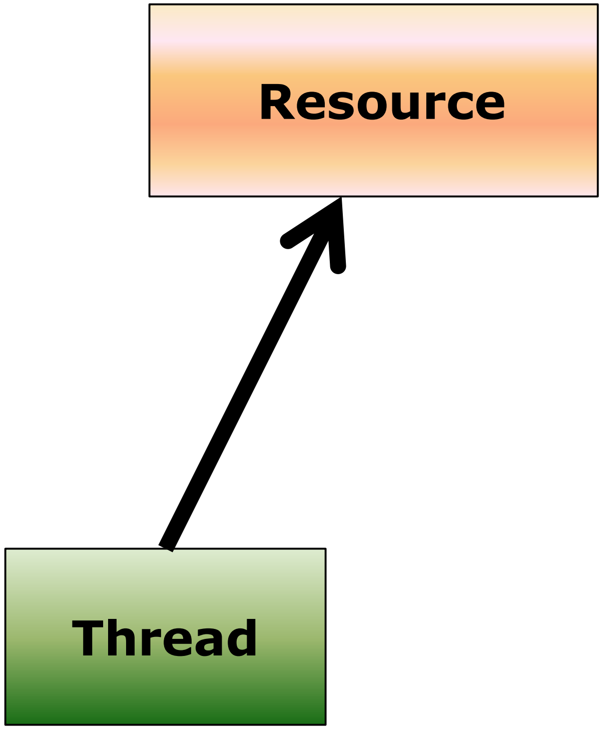
```
Thread();

Thread(Runnable);

Thread(String);
```

Thread class

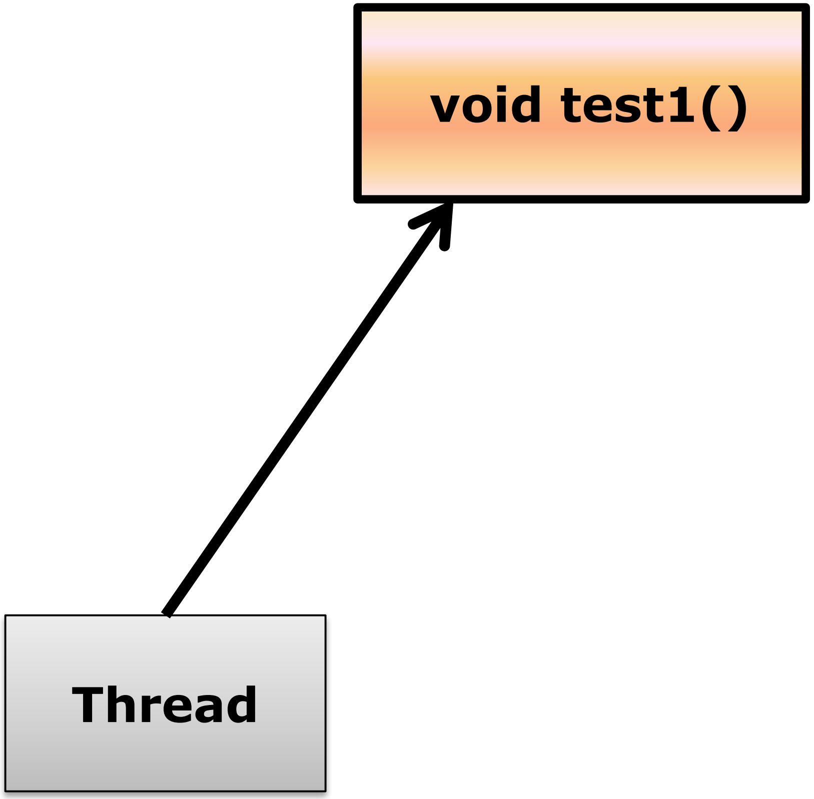
```
Thread(Runnable)

class A implements Runnable
{

}
```

```
                    A ob=new A();

                    Thread t1=new Thread(ob);
```

Synchronization



Synchronization



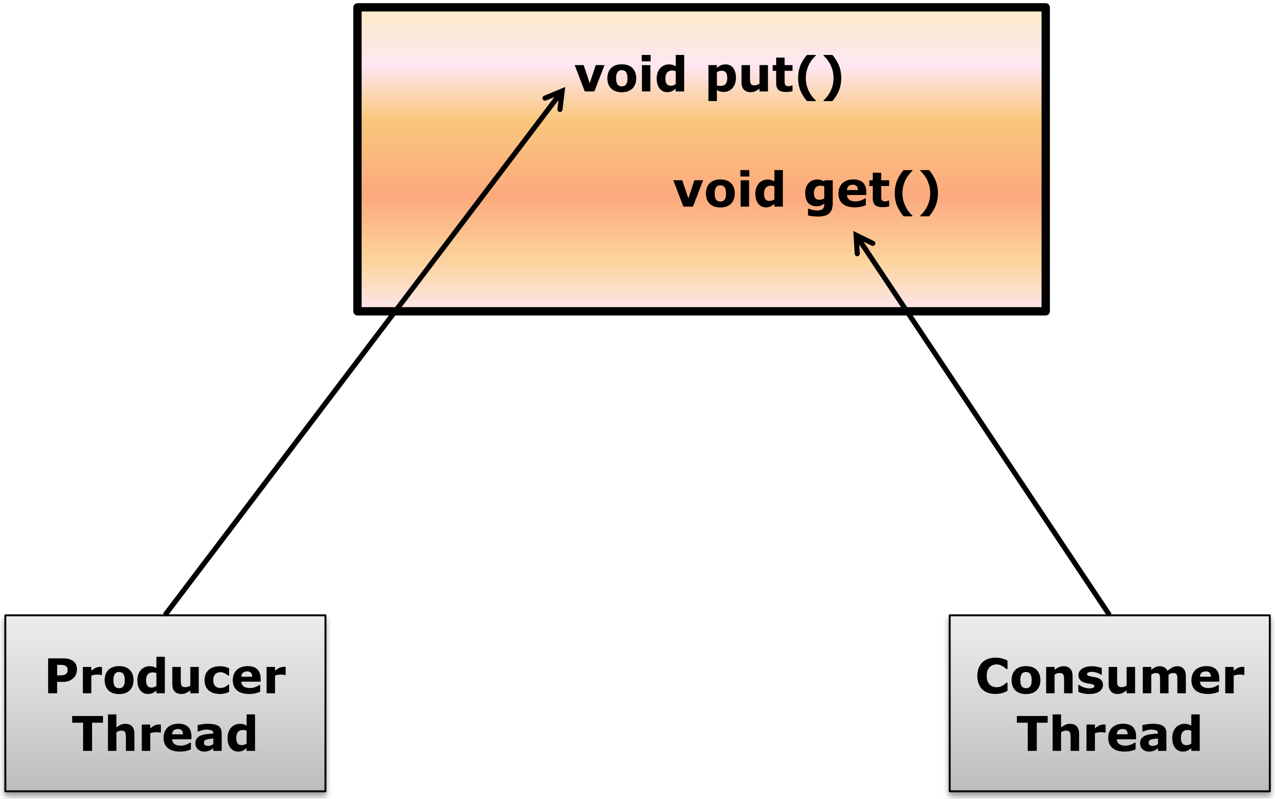
Synchronized Method

```
synchronized void test1()
{

        10000 Lines

 }
```

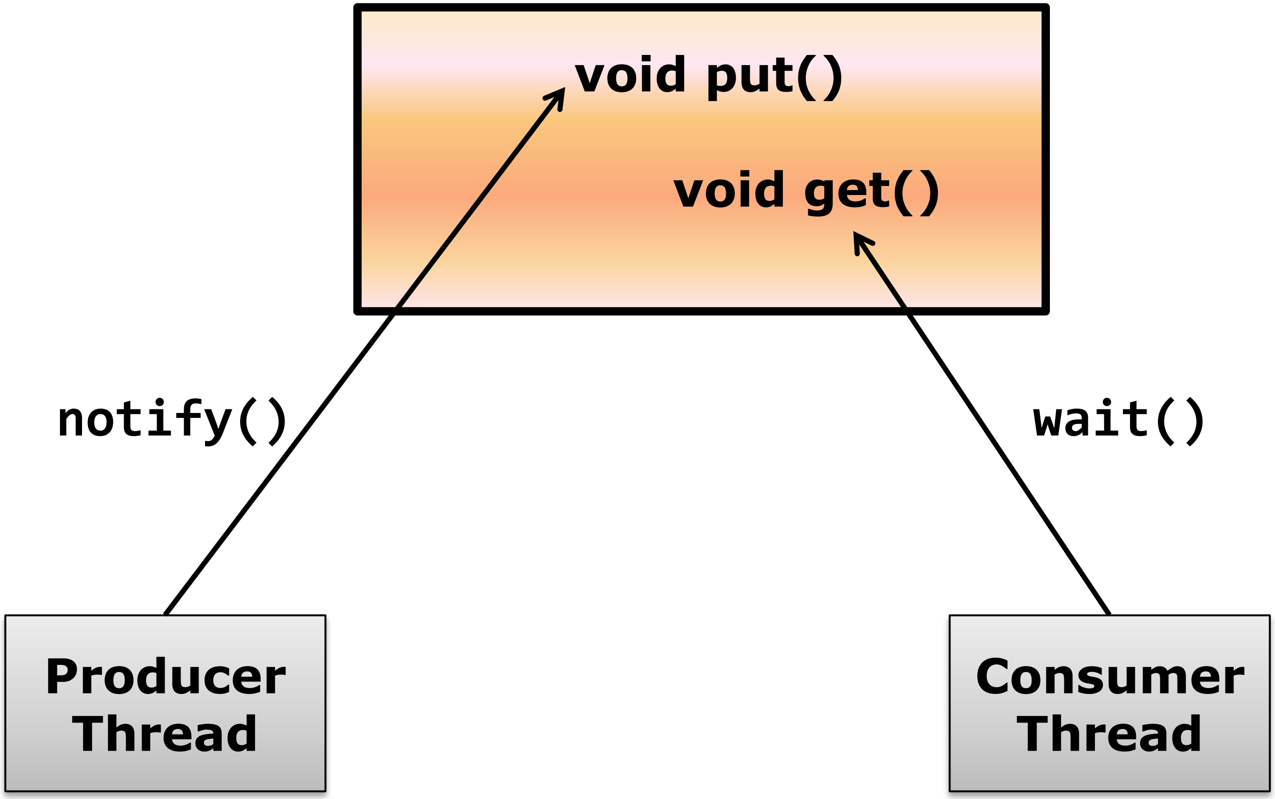
Synchronized Block

```
void test1()
{
    9000
     synchronized(this)
     {
            1000;
     }

 }
```

Synchronization



Synchronization



Thread Synchronization

- Monitors
  - Object with **`synchronized`** methods
    - Any object can be a monitor
  - Methods declared **`synchronized`**
    - **`public synchronized int myMethod( int x )`**
    - Only one thread can execute a **`synchronized method`** at a time
      - *Obtaining the lock* and *locking* an object
    - If multiple **`synchronized`** methods, only one may be active
  - Java also has **`synchronized`** blocks of code

Thread Synchronization

- Thread may decide it cannot proceed
  - May voluntarily call **`wait`** while accessing a **`synchronized`** method
    - Removes thread from contention for monitor object and processor
    - Thread in waiting state
  - Other threads try to enter monitor object
    - Suppose condition first thread needs has now been met
    - Can call **`notify`** to tell a single waiting thread to enter ready state
    - **`notifyAll`** - tells all waiting threads to enter ready state

Daemon Threads

- Daemon threads
  - Threads that run for benefit of other threads
    - E.g., garbage collector
  - Run in background
    - Use processor time that would otherwise go to waste
  - Unlike normal threads, do not prevent a program from terminating - when only daemon threads remain, program exits

Synchonized blocks

- Synchronized blocks of code

```
synchronized( monitorObject ){
    ...
}
```

  - **monitorObject**- Object to be locked while thread executes block of code – Why?
- Suspending threads
  - In earlier versions of Java, there were methods to stop/suspend/resume threads
    - Why have these methods been deprecated?
    - Dangerous, can lead to deadlock
  - Instead, use **wait** and **notify**
    - **wait** causes current thread to release ownership of a monitor until another thread invokes the **notify** or **notifyAll** method
    - Why is this technique safer?