unconditionally

```
cont = true;
con
  while cont do
    */nix*/ end;
  cont = false;
end;
```

Nebenläufigkeit (con): Programmteile werden nebeneinander, gleichzeitig ausgeführt
X= 0, y = 0
Con x = x+1; y=y+1; end;
Z=x+y // wird erst ausgeführt wenn alle sachen im con-statement abgearbeitet sind

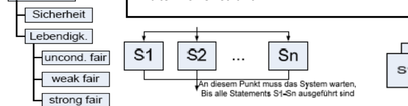
Weak fair:

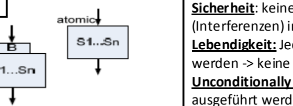
```
cont = true;
con
  while cont do
    try = true;
    */nix*/
    try = false;
  end;
  await try then cont
      = false; end;
end;
```

Strong fair:

```
cont = true;
con
  while cont do
    try = true;
    Thread.Sleep(0);
    try = false;
  end;
  await try then cont
        = false; end;
end;
```

Synchronisation (lock): Thread reserviert einen Codebereich für sich / gesetzt beim Eintritt und freigegeben beim Austritt aus dem Codeabschnitt / Falls schon besetzt, muss Thread warten, bis wieder freigegeben / mit lock() erstellen, es können nur Codeblöcke blockiert werden, Methoden gehen nicht // lock entspricht dem Atomic, lock() muss ein Objekt als Parameter besitzen Bsp:
Int sum;
Static object locker = new objekt();
// thread1 berechnet Summe…
Lock(locker){ sum = 2; } …
// Thread2 gibt Summe aus…
Lock(locker){ Console.WriteLine(sum); }
// Sum ist eine Klassenvariable und kann von mehreren Threads überschrieben werden. Im lock() Konstrukt ist dies gesperrt -> atomic konstrukt





Atomare Teile: ist atomar, wenn kein Zwischenschritt, kein Zwischenergebnis und kein Zwischenzustand in anderen parallel abarbeitenden Programmen sichtbar ist.
X=y=0; § Con § Atomic z=x+y; end;
Atomic x=1; y=2; end; § end; -> wird zwar parallel ausgeführt, jedoch ist nicht sicher, dass x=1, y=2 vor der Summation initialisiert werden -> evtl. 0 als Resultat

HTTP Response Codes:
1xx: informelle Meldungen: Request erhalten, Bearbeitung wird durchgeführt
2xx: Erfolg: Request wurde erfolgreich erhalten, verstanden und angenommen
3xx: Weiterleiten: Weitere Aktionen müssen eingeleitet werden, damit ein Request vollständig bearbeitet werden kann
4xx: Clientfehler: Der Request enthält ungültigen Syntax oder kann nicht bearbeitet werden.
5xx: Serverfehler: Der Server kann einen gültigen Request nicht bearbeiten
404 = „File not Found“
403 = „Forbidden“

Explizite Synchronisation (await): Warten bis Initialisierung abgeschlossen ist, bis Resultat gerechnet wurde Abhängigkeit der Nebenläufe (await per Def. Atomar)
Notation: atomic await B [then s1…sn] end; end;
Es wird solange gewartet, bis B (boolean) true wird, danach wird Programm weitergeführt. Bedingung B muss in einer parallelen ausführung (z.B. thread) true werden
X=y=0; § Con § Await (x>0 && y>0) then z=x+y; end;
X=1; § Y=2; § End;

Threads starten (ohne parameter):
Thread t = new Thread(new ThreadStart(Go));
// Go = irgendeine methode
t.Start(); // starten des threads (der Methode Go() in Form eines Threads)
Go(); // normaler Aufruf der Methode Go()
Threads starten (mit Parameter): (Lange Schreibweise)
Übergeben Parameter muss vom Typ object sein
Thread t = new Thread(new ParametrizedThreadStart(Go)),
t.Start(true); // thread Start
Go(false); // normaler Aufruf der Methode Go(boolean)
Kurze Schreibweise: Thread t = new Thread(Go);
t.Start(true);
static void Go(object obj){ bool upper = (bool) obj; }
Prioritäten: Den Threads können Prioritäten zugeteilt werden, abgearbeitet -> t.Priority = ThreadPriority.Lowest

Threads: CLR = Common Language Runtime
Blocked → ready = I/O fertig, wake up, join fertig
Running -> blocked = I/O Operation, Sleep, join
New: Thread Objekt erzeugt, aber noch nicht gestartet / **Ready:** gestartet, lockaler Speicher(stack) zugeteilt, wartet auf Zuweisung vom Prozessor / **Running:** führt Anweisungen auf dem Prozessor aus / **Blocked:** Thread muss warten bis Bedingung erfüllt ist / **Stopped:** Thread Objekt existiert nicht mehr und kann vom Garbage Collector entfernt werden /
Threads können untereinander interagieren -> evtl. müssen sie aufeinander warten / Nur 1 Thread im running-Zustand möglich pro Core

Threads Beenden: Thread ist beendet, wenn:
- Thread Methode beendet ohne Fehler,
- in Thread Methode tritt eine Exception auf -> z.B. Division durch Null -> Abbruch,
- Thread Abbruch von Aussen t.Abort();
Interrupts werden von Threads nur bearbeitet, wenn er im blocked oder wait Zustand ist (aufwecken aus dem blocked Zustand), Bei Aufruf im Running Mode läuft er einfach weiter / Exceptions müssen selber ausprogrammiert werden

```
Try { Thread.Sleep(1); }
Catch(ThreadInterruptedException){
Console.WriteLine(„blabla“); }
Finally { …. }
```

// überprüfen ob thread im blocked Zustand ist

```
if(t.ThreadState&
ThreadState.WaitSleepJoin) >0
```

t.Abort(): wirft zur Laufzeit des Threads eine AbortException -> Methode wird ordnungs-gemäss beendet. Z.B. werden offene Ressourcen geschlossen.
t.ResetAbort(): Bei t.Abort() wird ein Flag gesetzt und beim nächsten Eintreffen des Threads in den Running Zustand wird die Exception geworfen. Will man den Thread nicht sterben lassen, kann in der catch{} mit ResetAbort() der Thread wieder aufgeweckt werden.
Try{ while(true);….}
Catch(ThreadAbortException){
Thread.ResetAbort(); }

Allgemeine Aussagen:
Threadsicherheit: Zugriffe mehrerer Threads auf gemeinsamen Speicher sind kritisch!
Monitor: Ein Monitor ist ein Konstrukt, das kontrollierten Zugriff auf Datenelemente erlaubt.
Synchronisation: Der Objects lock-pool und der Objects wait-pool müssen immer zum selben Objekt gehören, ist dies nicht der Fall, führt es während der Laufzeit zu einer „System.Threading. SynchronizationLockException“
Join: t.join() bewirkt, das der aufrufende Thread wartet (Zustand blockiert), bis t seine Arbeit abgeschlossen hat und terminiert.
Wait & Pulse: Wait und Pulse dürfen nur innerhalb eines lock-Konstrukts aufgerufen werden!

Sicherheit: keine Verklemmungen(Deadlock) durch gegenseitige Zugriffe (Interferenzen) im kritischen Bereich
Lebendigkeit: Jeder Programmteil kriegt eine faire Chance ausgeführt zu werden -> keine Livelock! (Starvation -> Hungern)
Unconditionally fair: es gibt keine Bedingung, so dass beide Statements ausgeführt werden / cont wird bei einem Taskwechsel auf jeden Fall auf false gesetzt -> while terminiert
Weak fair: es ist unsicher, ob beide ausgeführt werden / es kann sein, dass cont nie false wird und die Schleife nicht terminiert, da unter Umständen try beim Taskwechsel immer false ist → Zufall
Strong fair: es ist garantiert, dass alle zum Zug kommen / es ist ein Taskwechsel garantiert, jeder Thread kriegt eine faire chance / es wird ein Taskwechsel zwischen try = true und try = false erzwungen mittels await … oder Thread.Sleep(0);

Mutex (wechselseitiger Ausschluss): Nebenläufige Prozesse / Threads können nicht gleichzeitig auf Daten zugreifen / Jedes Mutex ist ein String
bin.P(): await b then b=false; end;
bin.V(): atomic await !b then b=true; end;

```
Im folgenden Beispiel wird ein Mutex
erzeugt, das beim Start der App während
5s geprüft wird, ob es frei geschaltet
ist. Ist dies nicht der Fall wird die
App wieder beendet.
class MutexTest {
 static Mutex mutex = new
  Mutex(false, "NameDerSoftware");
 static void Main(){
  if (!mutex.WaitOne(TimeSpan.FromSeconds(5),
     false)){
    Console.WriteLine("Another is running ");
    return; }
  try{
   Console.WriteLine("Running ");
   Console.ReadLine();}
```

Anwendung von Semaphoren im Bounded Buffer:
Eine sehr häufige Aufgabenstellung in der nebenläufigen Programmierung ist, dass n Produzenten Daten erzeugen, die an m Konsumenten weitergegeben werden sollen. Zur Weitergabe dient ein Puffer, der in der Lage sein soll, k Daten zu speichern. Dabei soll m>=1, n>=1 und k>=1 sein. Falls der Puffer voll ist, sollen die Produzenten warten, bis wieder Platz ist, und falls der Puffer leer ist, sollen die Konsumenten warten, bis wieder Daten vorhanden sind. Eine Synchronisation muss sicherstellen, dass der Produzent wartet, falls der Puffer voll ist, und der Konsument wartet, falls der Puffer leer ist. Damit ergibt sich folgender Ansatz:

```
global
  x = 0; k = bufferSize;
 Producer(i):
  while (true) {
   /* Daten erzeugen */
   await x < k then
    /*Data in Puffer schreiben*/
    x = x + 1;
   end;}
 Consumer(i):
  while (true) {
   await x >= 1 then
    /*Data aus Puffer lesen*/
    x = x - 1; end;
  /* Daten verbrauchen */}
```

Sequentieller Server:
(1) Warte, bis eine Anfrage (request) von einem Client eintrifft
(2) Verarbeite die Anfrage des Client
(3) Sende das Ergebnis der Anfrage an den Client zurück, der diese gestellt hat
(4) Gehe zurück zu Schritt (1)

Gleichzeitiges vergleichen der x-Ressource von Produzent und Konsument ist nicht möglich -> Produzent und Konsument können nicht gleichzeitig auf den Puffer zugreifen -> Eine Abhilfe schafft die Verwendung von zwei Zählern, bzw. empty und full

```
empty = Buffersize;  full = 0;
con
 Producer(1); ---; Producer(n);
 Consumer(1); ---; Consumer(m)
end;
 Producer(i):
  while (true) {
   /* Daten erzeugen */
   await empty >= 1 then empty--;
    /* Daten in Puffer schreiben */
    atomic full++;
   end;}
 Consumer(i):
  while (true) {
   await full >= 1 then full--;
    /* Daten aus Puffer lesen */
    Atomic empty++; end;
  /* Daten verbrauchen */}
```

Semaphore: Allgemeines Konzept / Anzahl Threads, welche Zugriff auf einen kritischen Abschnitt haben sollen, können bestimmt werden / zwei Operationen, sema.P = Eintritt, sema.V = Verlassen /
sema.P(): await s>0 then s= s-1; end;
sema.V(): atomic s=s+1; end;
Semaphore(1,1) ist wie ein lock-Konstrukt / Static Semaphore s = new Semaphore(1,3);
// die erste Zahl (=avaliable) 1 besagt, wieviele Threads zu Beginn eintreten können, ohne, dass zuerst ein s.Release() gemacht werden muss-
Bei 0 müssten also alle Threads am Anfang mal warten! Kann maximal den Wert der zweiten Zahl (=capacity) annehmen, die besagt, wieviele Threads tatsächlich gleichzeitig im kritischen Abschnitt drin sein dürfen. (capacity muss min. 1 sein sonst wirft er ein Exception.
/// s.WaitOne() lässt den Thread in den kritischen Abschnitt eintreten
`s. Release();` lässt den Thread aus kritischen Abschnitt austreten und erhöht freier Plätze um 1// `static Semaphore s = new Semaphore(0,3); //`
while (true) (`s.WaitOne();`// wartet, bis Einlasssignal kommt // kritischer Code, dann `s. Release();`// Beim Verlassen Slot wieder freigeben, sonst wäre nach 3 Durchläufen die capacity erreicht und es kommt niemand mehr rein

Wait Handle: sobald wh.Set() -> wh im Zustand aktiv, bei **AutoResetEvent** automatisch zurück in Zustand inaktiv, nachdem 1 Thread aktiviert, bei **ManualResetEvent** wird der Zustand nicht automatisch zurückgesetzt bis wh.Reset() Somit werden mehrer Threads geweckt

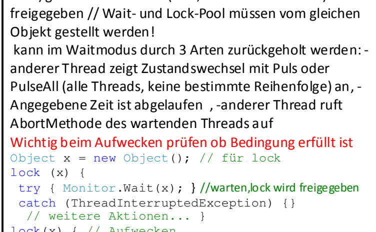
```
class BasicWaitHandle {
  static EventWaitHandle wh = new
        AutoResetEvent(false);
  static void Main(){
   new Thread(Waiter).Start();
   Thread.Sleep(1000);
   wh.Set();}
  static void Waiter(){
   Console.WriteLine("Waiting...");
   wh.WaitOne();//warten auf wh.set
   Console.WriteLine("Notified");}}
```

Einfache Blockierung:

```
static void Main(){
  long sum = 0;
  bool fertig = false;
  Thread t = new Thread(delegate(){
   for (int i = 0; i <= 1000; i++){
    for (int j = 0; j < 100000; j++) ;
      sum += i;
   }
   fertig = true;});
  t.Start();
  while (!fertig) ; // Version 1 switcht immer
zwischen Main thread und t hin und her
  Thread.Sleep(1000); // Version 2 unsicher
  t.Join();// Version 3 beste version
  Console.WriteLine("Summe = " + sum);}
```

Wait und Pulse: Idee: Die Threads warten an einem Monitorobjekt, bis dieses einen Impuls erhält, um einen oder mehrere Threads frei zu schalten // Wait und Pulse nur innerhalb eines lock-Bereiches aufrufen!! -> bei Waitaufruf wird Thread in Wartezustand (Objects-Wait-Pool) geschickt (lock, kritischer Abschnitt) freigegeben // Wait- und Lock-Pool müssen vom gleichen Objekt gestellt werden!
kann im Waitmodus durch 3 Arten zurückgeholt werden: -anderer Thread zeigt Zustandswechsel mit Puls oder PulseAll (alle Threads, keine bestimmte Reihenfolge) an, -Angegebene Zeit ist abgelaufen , -anderer Thread ruft AbortMethode des wartenden Threads auf
Wichtig beim Aufwecken prüfen ob Bedingung erfüllt ist

```
Object x = new Object(); // für lock
lock (x) {
 try { Monitor.Wait(x); } //warten,lock wird freigegeben
 catch (ThreadInterruptedException) {}
  // weitere Aktionen... }
lock(x) { // Aufwecken
 Monitor.Pulse(x); }
```



Bounded Buffer: Produzenten erstellen Daten, die in einem Buffer zwischengespeichert werden / Die Konsumenten holen die Daten aus dem Buffer / ist Buffer voll, warten die Produzenten / ist Buffer lehr, müssen die Konsumenten warten / Buffer würde mit einem Längenindikator funktionieren / Zugriffe auf „await“ Anweisung / innerhalb dieser Anweisung sind die Zugriffe atomar! / Effizienter mit zwei Indikatoren -> empty = BufferSize, full = 0 ->
Implementierung mit RingBufferArray:

```
class BoundedBufferWithSemaphor : IQueue{
protected Semaphore empty;
protected Semaphore full;
protected RingBufferArray buf;
public BoundedBufferWithSemaphor(int
size) {
   buf = new RingBufferArray(size);
   empty = new Semaphore(size,size);
   full = new Semaphore(0,size);
}
public void Enqueue(Object x){
   empty.WaitOne();
   buf.Put(x);
   full.Release();}
public Object Dequeue(){
   full.WaitOne();
   Object x = buf.Get();
   empty.Release();
   return x;}}

class RingBufferArray{
  protected Object[] array;
  protected int front = 0;
  protected int rear = 0;
  private Object putLock = new Object();
  private Object getLock = new Object();

  public RingBufferArray(int n) {
     array = new Object[n];}
  public void Put(Object x) {
     lock (putLock) {
          array[rear] = x;
  rear = (rear + 1) % array.GetLength(0);
     }
  }
  public Object Get() {
     lock (getLock) {
       Object x = array[front];
       array[front] = null;
front = (front + 1) % array.GetLength(0);
       return x;}}}
```

Streams realisieren generell **3 elementare Operationen**:
-Dateninformationen müssen in einen Stream geschrieben werden
-Aus dem Datenstrom muss gelesen werden
-Wahlfreier Zugriff: „Nur von A bis nach B lesen“
Netzwerk TCP-Socket Verbindung(nur Abfrage) einer bestimmten Adresse `using`

```
System.Net.Sockets; using System.IO;
public static void Main() {
  TcpClient client = new TcpClient("192.53.103.103", 13);
  StreamReader inStream = new StreamReader(client.GetStream(
  ));
  Console.WriteLine(inStream.ReadLine());
  client.Close();}
```

Schreiben in eine Datei mit FileStream: public static void Main() {

```
  try {FileStream fs = new FileStream("daten.txt",  FileMode.Create);
   StreamWriter sw = new StreamWriter(fs);
   string[] text = { "Titel", "Köln", "4711" };
   for (int i = 0; i < text.Length; i++){sw.WriteLine(text[i]);}
   sw.Close();
   Console.WriteLine("fertig.");}
 catch (Exception e){Console.WriteLine(e.ToString());} }
```

Lesen aus einer Datei public static void Main() {

```
  try { using (StreamReader sr = new StreamReader("daten.txt")){
     string line;
     while ((line = sr.ReadLine()) != null) {Console.WriteLine(line);}}}
 catch (Exception e) {Console.WriteLine(e);}}}
```

Streams: Base-Streams & Pass-Through-Streams
Base Streams lesen/speichern
Pass-Through-Streams ergänzen die Base Streams, eigene Zusammenstellungen möglich
I/O-Streams: Binäre (BinaryWriter,BinaryReader): Arbeitet ohne Umwandlung der Binärdarstellung
Zeichen/textorientiert (StreamReader/Writer, StringReader/Writer): wandelt Bin. In Textdarstellung
Base-Streams: TextReader, FileStream, NetworkStream, MemoryStream

Wichtig Flush() bei Writing Operationen nicht vergessen!!!

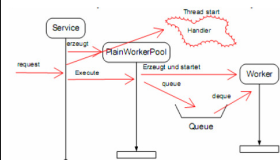
AbstractServer: Ein konkreter Server kann, bzw. muss die folgenden drei Methoden implementieren:

`CreateExecutor` – Methode zur Erzeugung eines Executors. Mit dem Executor hat man die Möglichkeit den Server skalierbar zu machen. Je nach Bedürfnis kann der Executor mit verschiednen Buffern und unterschiedlicher Anzahl von ausführenden Threads initialisiert werden. Mit der Standardimplementierung wird ein einfacher Thread-Executor ohne Buffer erzeugt.

`CreateServerSocket` – Methode zur Erzeugung eines Serversockets. Diese Methode eröffnet die Möglichkeit einen Server Socket auf eine spezifische unterschiedliche Art zu erzeugen. Mit der Standardimplementierung wird ein `TcpListener`-Objekt am definierten Port erzeugt.

`CreateHandler` – Methode zur Erzeugung eines Handlers. Der Handler bestimmt die Funktion des Servers und muss daher mit durch eine konkrete Handler-Klasse implementiert werden.

Executor mit Worker Threads:



Beispiel eines Executors mit Worker Threads

```
interface IQueue {
 void Enqueue(Object x);
 Object Dequeue();}
interface IExecutor {
 void Execute(ThreadStart threadStart);}
interface ICallbackHandler {
 void Handle(Object res);}
class TestWorkerPoolExec {
 public static void Main() {
  PlainWorkerPool executor = new PlainWorkerPool( // Executor
  new BoundedBufferWithSemaphor(100), // Queue
  10); // Anzahl Workers
  Service service = new Service(executor);
  for (int i = 0; i < 10; i++) { service.Request();
}}}----------------------------------------
class PlainWorkerPool : IExecutor {
 protected IQueue workQueue;
 public PlainWorkerPool(IQueue workQueue, int
nWorkers){
  this.workQueue = workQueue;
  for (int i = 0; i < nWorkers; ++i) {activate();}}
 public void Execute(ThreadStart threadStart) {
  workQueue.Enqueue(threadStart);}
 protected void activate() {
  Thread runLoop = new Thread(delegate() {
   while(true) {
    ThreadStart threadStart = (ThreadStart)workQueue.Dequeue();
    threadStart.Invoke();}});
  runLoop.Start();
 }}-------------------------------------
class Service : ICallbackHandler {
 protected IExecutor executor;
 public Service(IExecutor executor) {
  this.executor = executor;}
 public void Request() {
  Console.WriteLine("Die Aufgabe und Referenz auf sich selbst");
  Console.WriteLine("an den Handler übergeben und starten...");
  Handler handler = new Handler(this);
  executor.Execute(handler.Do);
  for(int i = 0; i < 5; i++) {
   Console.WriteLine("weitere Aufgaben abarbeite");
   Thread.Sleep(300);}}
 public void Handle(Object res) {
  Console.WriteLine("Resultat zur Verfügung: " + res.ToString()); }
}--------------------------------------
class Handler {
 ICallbackHandler service;
 public Handler(ICallbackHandler service) {
  this.service = service;}
 public void Do() {
  Console.WriteLine("Die Aufgabe lösen...");
  Thread.Sleep(1000);
  Console.WriteLine("Aufgabe beendet, Service benachrichtigen");
  service.Handle("Fertig.");}
}--------------------------------------
class BoundedBufferWithSemaphor : IQueue {
 protected Semaphore empty;
 protected Semaphore full;
 protected RingBufferArray buf;
 public BoundedBufferWithSemaphor(int size) {
  buf = new RingBufferArray(size);
  empty = new Semaphore(size,size);
  full = new Semaphore(0,size);}
 public void Enqueue(Object x) {
  empty.WaitOne();
  buf.Put(x); full.Release();}
 public Object Dequeue() {
  full.WaitOne();
  Object x = buf.Get(); empty.Release();
  return x;}
}--------------------------------------
class RingBufferArray {
 protected Object[] array;
 protected int front = 0;
 protected int rear = 0;
 private Object putLock = new Object();
 private Object getLock = new Object();
 public RingBufferArray(int n) {
  array = new Object[n];}
 public void Put(Object x) {
  lock (putLock) {
   array[rear] = x;
   rear = (rear + 1) % array.GetLength(0);}}
 public Object Get() {
  lock (getLock) {
   Object x = array[front];
   array[front] = null;
   front = (front + 1) % array.GetLength(0);
   return x;}}}
```

BlueTooth: Service Starten

```
using HTA.BlueTooth;
if (BluetoothRadio.IsSupported) //wenn BT-Stick da
{BluetoothRadio.PrimaryRadio.Name = "HTARobot";
Guid serviceId = BluetoothServiceList.HTA_Robot;
BluetoothService service = new BluetoothService();
service.CreateService(serviceId); }
```

Verbindung annehmen

```
BluetoothClient client;
System.Net.Sockets.NetworkStream ns;
while (true) {client = service.WaitForConnection();
Console.WriteLine("Incoming connection");
ns = client.GetStream(); StreamWriter sw = new
StreamWriter(ns); sw.WriteLine("Hello from
Server!"); sw.Flush(); client.Close();}
```

Erreichbare Geräte suchen

```
Guid service = BluetoothServiceList.HTA_Robot;
BluetoothDiscovery discovery = new
BluetoothDiscovery();
BluetoothDeviceCollection bdc =
discovery.DiscoverDevices(12, false, false, true);
foreach (BluetoothDevice device in bdc) {
   if (device.HasService(service)) {
            Console.WriteLine(device.Name + ";"
            +device.DeviceAddress); }}
```

Zu Gerät verbinden

```
if (device.HasService(service)) {BluetoothClient bc
= device.Connect(service); StreamReader sr = new
StreamReader(bc.GetStream()); Console.WriteLine
("Message received : "); Console.WriteLine
(sr.ReadLine()); bc.Close();}
else {Console.WriteLine("Required service does not
exist on target");}
```

Lost Signal:

Reihenfolge der Thread wann welcher gestartet wird ist wichtig. Wenn Arbeiten freigegeben werden bevor Worker exisitieren, dann gehen dies Released Meldungen verloren -> darum Semaphore verwenden welcher mitzählt.

Asynch mit Callback

```
interface ICallbackHandler {
 void Handle(Object res);
    class Service : ICallbackHandler
    {
        public void Request()
        {
            Handler handler = new Handler(this);
            new Thread(handler.Do);
        }
        public void Handle(object res) {/*result*/}
    }
    class Handler
    {
        ICallbackHandler service;
        public void Do()
        {/*Aufgabe*/ service.Handle(result);}
    }
```

Asynch mit Polling

```
Thread worker = new Thread(handler.Do);
worker.Start(); while(worker.IsAlive){} //prüfen ob
Thread noch läuft
```

Await Implementation

Arbeit auf einen Tester und einen Waiter aufteilen

```
class Await
        {
         Boolean cond=false;
         public void Await(Boolean b, Runnable s){
                lock(this){
                    cond = b;
                    while(!cond)
                    {
                        Monitor.Wait(this);
                    }
                    if(s!=null){s();}
                }
            }
            public void Test(Boolean b){
                lock(this){
                    cond = b;
                    if(cond)
                    {
                        Monitor.Pulse(this);
                    }}}
        }
```

Socket-Prinzip-Verbindungen: -> using System.Net.Sockets / Socket = Datenendpunkt eines Prozesses / Prozesse benutzen die Socket-Verbindung um untereinander zu kommunizieren, da sie keinen gemeinsamen Speicher haben / Socketkommunikation = Interprozesskommunikation, zwei oder mehrere Prozesse kommunizieren miteinander.

Was ein Socket beherrschen muss: - Sich mit einem anderen Socket verbinden, - Daten senden und empfangen können, - Verbindung wieder ordnungsgemäss beenden, -Einen Port binden, - An einem Port auf eine Verbindungsanfrage hören, - Verbindungsanfrage am Port akzeptieren können // Übertragungsprotokoll nicht sichtbar / Einfaches Lesen und Schreiben (Technik dahinter interessiert nicht) / Socket Kommunikation zwischen 2 oder mehreren Prozessen (PC-PC,Programm-Speicher) / Alle sprechen gleiche „Sprache" (meistens TCP/IP Protokoll) /

Die Sockets werden direkt mit der Klasse NetworkStream verknüpft. Ein TcpListener wartet auf einem vorbestimmten Port auf eine Anfrage eines Clients. Die Methode AcceptTcpClient() des TcpListeners ist blockierend und liefert ein Objekt vom Typ Client. Die Methode Accept() weist dasselbe Verhalten auf, liefert jedoch ein Objekt vom Typ Socket. / Prozess Ports > 1024, Dienste Ports <1024 /

TCP-Client: TcpClient client = new TcpClient(„hostname",Portnummer) / ein TCP Objekt braucht immer einen Hostnamen (kann auch IPadresse sein) und eine Portnummer (int)

Socket-Eigenschaften: -Herstellung einer Verbindung auf einen Prozess oder einen Dienst (Portgrössen siehe oben) / Dienst ist z.B. ein Webserver / Prozess ist z.B. Verbindung und Fernsteuerung eines PC's (RemoteDesktopPort >1024) socket.RemoteEndPoint, .LocalEndpoint, .ProtocolType

Einfache TCP Verbindung: Verbindet auf die IP 1.1.1.1, Port 80

```
public static void Main(string[] args) {
 try {
    TcpClient theClient = theClient = new
       TcpClient("1.1.1.1", 80);
    Console.WriteLine("Yesssss! Verbindung
       erfolgreich!!"); }
 catch (Exception e){ Console.WriteLine("Zonk!");}}
```

ServerSocket (Aufbau auf Server, z.B. Time-Server):

Der ServerSocket ist die Implementierung eines Servers, welcher auf eine Verbindung wartet. Dazu wird ein TcpListener Objekt für einen bestimmten Port erzeugt & gestartet. Danach wartet die Methode AcceptTCPClient auf eine Verbindung. -> AcceptTcpClient wartet solange bis eine Verbindung erstellt wird! Kann alles blockieren!

```
TcpListener listen = new TcpListener(13);
// server wartet auf port 13 auf einen Client
listen.Start();
while (true) {
  TcpClient client = listen.AcceptTcpClient();
  // warten, bis jemand Verbindet, dann generiert er einen
  // TcpClient, der mit dem anderen kommuniziert
  Console.WriteLine("Verbindung zu " +
       client.Client.RemoteEndPoint);
  StreamWriter sw = new
       StreamWriter(client.GetStream());
  //getstream um zu wissen wohin man schreiben muss
  sw.Write(DateTime.Now.ToString());
  sw.Flush();
  client.Close();}
```

Wichtig: bei diesem Beispiel kann nur immer eine Verbindung erstellt werden. Für mehrere gleichzeitige Verbindungen sollte wiederum ein PlainworkerPool eingesetzt werden!

```
public interface IExecutor {
  void Execute(ThreadStart threadStart);
}----------------------------------------
public class PlainThreadExecutor : IExecutor {
 public void Execute(ThreadStart threadStart){
  new Thread(threadStart).Start();
  // hier eine Queue implementieren werden,
}
}----------------------------------------
public abstract class AbstractServer {
 private Boolean running = true;
 public const int DEFAULTPORT = 4711;
 public const string DEFAULTHOST = "localhost";
 protected void Run(Object port){
  IExecutor executor = CreateExecutor();
  Socket listen = CreateServerSocket((int)port);
  while (running) {
   AbstractHandler handler = CreateHandler(listen.Accept());
    // wartet bis ein Client verbindet
   executor.Execute(handler.Run);
    // erzeugt Thread
  }
 }                                              [3]
 virtual protected IExecutor CreateExecutor(){
  return new PlainThreadExecutor();
 }
 protected Socket CreateServerSocket(int port){
  IPAddress ipAddress =
Dns.GetHostEntry(host).AdressList[0];
  TcpListener listener = new
TcpListener(ipAddress, port);
  listener.Start();
  return listener.Server;
 }
 public void Start(){
  this.Start(0);
 }                         [1]
 public void Start(int port){
  Thread server = new Thread(Run);
  if (port <= 0)                   [2]
  server.Start(DEFAULTPORT);
  else
  server.Start(port);
 }
  abstract protected AbstractHandler CreateHandler(Socket client);
}----------------------------------------
public abstract class AbstractHandler{
 private Socket client;
 public AbstractHandler(Socket client){
  this.client = client;
 }
 public void Run() {
  try {
   if (ReadRequest()){ CreateResponse(); }
   client.Close();
  }
  catch (Exception e){Console.Error.WriteLine(e);}
 }
 abstract protected Boolean ReadRequest();
 abstract protected void CreateResponse();
}

class HelloServer : AbstractServer {
 override protected AbstractHandler CreateHandler(Socket client) {
  return new HelloHandler(client);
 }
 public static void Main() {
  new HelloServer().Start();
  Console.WriteLine("HS auf {0} gest..",AbstractServer.DEFAULTPORT);
 }
}----------------------------------------
class HelloHandler : AbstractHandler {
 private StreamReader sr;
 private StreamWriter sw;
 public HelloHandler(Socket client) : base (client) {
  NetworkStream nws = new  NetworkStream(client,true);
  sr = new StreamReader(nws);
  sw = new StreamWriter(nws);
 }
 override protected Boolean ReadRequest() {
  String request = sr.ReadLine();
  return true;
 }
 override protected void CreateResponse() {
  sw.Write("Hallo");
  sw.Flush();
 }}
```

HTTP-Response:

```
sw.WriteLine("HTTP/1.0 200 OK");
sw.WriteLine("Server: Webiii");
sw.WriteLine("Content-Length: " +
output.Length);
sw.WriteLine("Content-Type: text/plain");
sw.WriteLine("");
sw.Write(output);
sw.Flush();
```

weitere Typen: text/html, image/gif, image/jpg

```
class HttpHandler : AbstractHandler {
 // Multipurpose Internet Mail Extensions (MIME)
 // Internet Erweiterungen für die Einbindung von Daten.
 private string[,] mimetypes = {
 {"html","text/html"}, {"htm", "text/html"},
{"jpg","image/jpeg"},{"jpeg","image/jpeg"}};
public HttpHandler(Socket client, int id) : base(client) {
  this.id = id;
  nws = new NetworkStream(client, true);
  sr = new StreamReader(nws);
  sw = new StreamWriter(nws);}
 override protected bool ReadRequest() {
  Console.WriteLine(id+". Incoming
request...");
  string headerline;
  ArrayList request = new ArrayList();
  //Request-Header-Zeilen lesen bis Leerzeile
  while ((headerline = sr.ReadLine()) != null && headerline != "") {
   request.Add(headerline);
   Console.WriteLine("< " + headerline);
  } // 1. Request-Zeile auf HTTP Methode GET untersuchen
  string[] tokens = ((string)request[0]).Split(new char[] {' '});
  if (tokens.Length >= 2 && tokens[0] ==
"GET") {
   // URL ermittlen
   if (tokens[1].StartsWith("/")){ url = tokens[1]; }
   // Start URL setzen
   if (url.EndsWith("/")) {url+= "index.html";
}  return true;}
  else {WriteError(400, "Bad Request");
  return false; }}
```

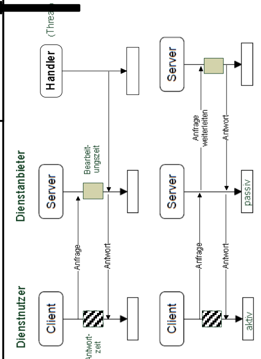
Client-Server Verfahren:

Am einfachsten erklärbar mit einem Webserver / mehrere Clients greifen zu / Der Server bietet Clients auf einem bekannten Port Port Dienste an. Das Client-Server-Pattern wird eingesetzt, wenn die Anzahl der Clients nicht bekannt ist, die Aufgaben nicht unabhängig voneinander sind, der Umfang der nachgefragten Leistungen variert und nicht genau bestimmbar ist und/oder ein Client z.T. Leistungen mehrerer Server in Anspruch nimmt. / Funktionsweise: Der AbstractServer erstellt in einer endlosen Schleife jeweils einen Handler, welcher den Client später bedienen wird. Dann wartet der Server mit einem TcpListener auf die Anfrage eines Clients. Sobald die Anfrage eintrifft, übergibt der Server die Run-Methode des Handlers einem Executor, welcher den Handler und dessen Aufgabe in einem separaten Thread ausführt. Der Handler verbindet den Client-Socket mit dem Input-/Outputstream, liest die Anfrage aus dem Inputstream und wertet diese aus. Dann wird eine entsprechende Antwort generiert und über den Outputstream an den Client gesendet.

Abstract Server: Ein Entwurfsmuster. Der Server erzeugt ein TcpListener Objekt und erhält von einem Client über das TcpListener Objekt einen Auftrag. Für die Erledigung des Auftrages wird ein Handler Objekt erzeugt, dem der Client Socket übergeben und in einem eigenen Thread gestartet wird./ **AbstractHandler**: Der Handler hat folgende Aufgaben zu erledigen: -Client-Socket mit dem Input- und Output Stream verbinden, -Anfrage aus dem Inputstream lesen und diese auswerten, -Eine entsprechende Antwort generieren und diese über den Output Stream an den Client senden

Beispiel eines Hallo-Servers:

- Wenn der Server gestartet wird, wird ein HalloServer Objekt erzeugt, welches wiederum im AbstractServer einen neuen Thread startet. Dieser Thread ist im Abstract Handler definiert.
- Der Thread im AbstractHandler führt zuerst die Methode ReadRequest() im HelloHandler aus. Diese wartet auf eine Eingabe des Benutzers. Ist diese erfolgt, geht das ganze zurück an den AbstractHandler Thread und jener führt die Methode CreateResponse() aus.
- Die Methode CreateResponse() im HelloHandler erstellt eine Antwort. Danach wird die Verbindung vom AbstractHandler geschlossen



Einfache Fileserver (paralleler Server)

Welchen Port müssen sie zum Start des Fileservers angeben? -> egal welcher (muss aber vorhanden sein / > 1024) Wie muss ein Browser den File-Server adressieren? -> http://localhost:4711/ Wie implementieren Sie ihren File-Server, so dass er skalierbar ist? -> Executor (im AbstractServer) überschreiben mit WorkerPool (im FileServer)