Resumen 1er parcial

Programación concurrente y en tiempo real

Programación concurrente

- Es el nombre dado a la notación y a las técnicas de programación para expresar el paralelismo potencial de un proceso y para resolver los problemas asociados de sincronización y comunicación
- La implementación de paralelismo es un tema que es esencialmente independiente de la programación concurrente

¿Para qué se necesita?

- Para modelar el paralelismo en el mundo real

- Virtualmente todos los sistemas de tiempo real son inherentemente concurrentes – los dispositivos operan en forma paralela en el mundo real

¿Qué es un sistema de tiempo real?

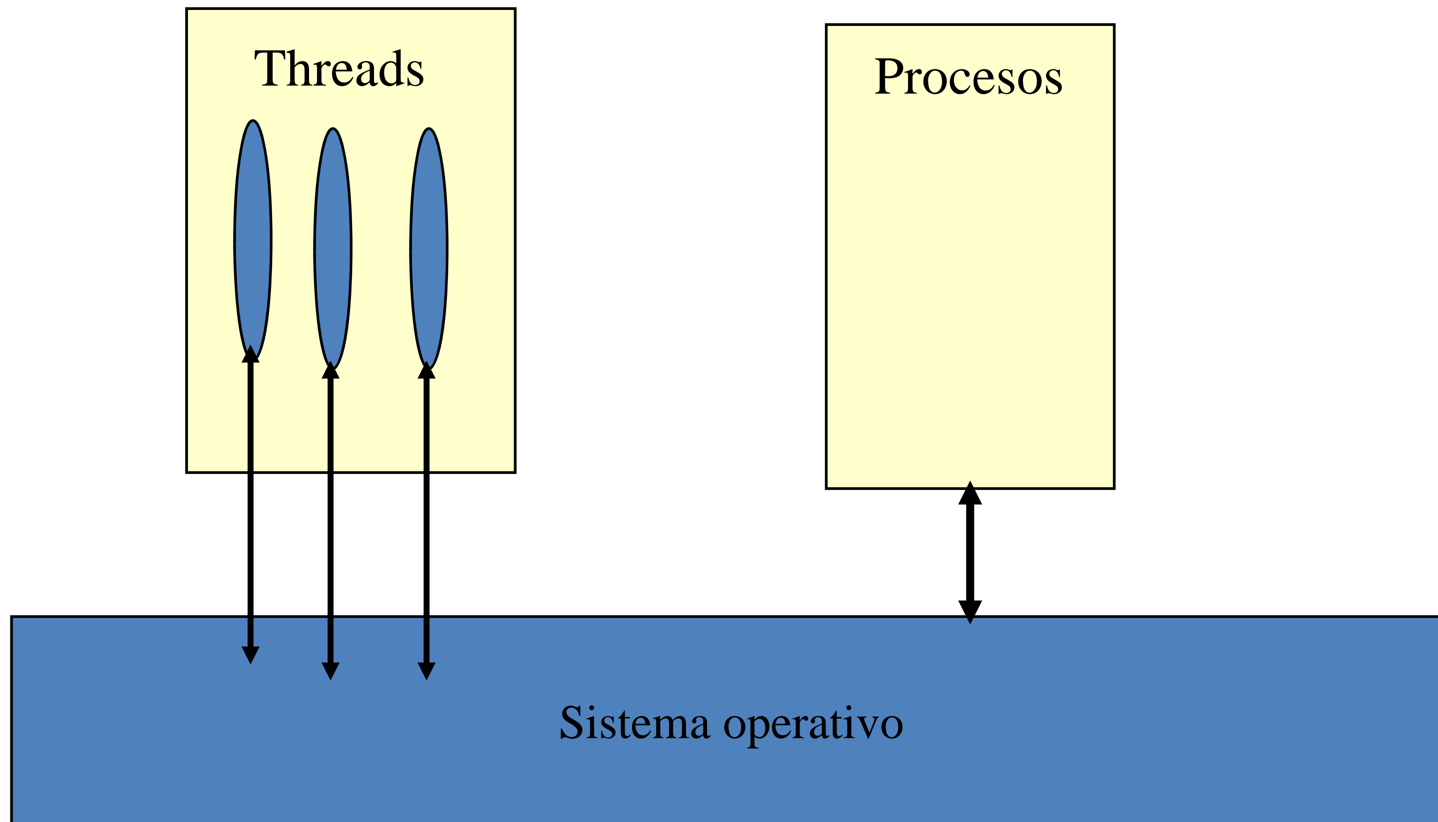
- Un sistema de tiempo real es cualquier sistema de procesamiento de información que responde a las entradas externas dentro de periodo especificado y finito
  - La operación correcta del sistema depende no solamente de la precisión del resultado sino del tiempo en el que fue entregado
  - La falla en responder a tiempo es tan mala como una respuesta equivocada!

Características de un STR

- Grande y complejo
- Control concurrente de componentes separados del sistema
- Facilidad para interactuar con hardware de propósito especial
- Extremadamente confiable y seguro
- Tiempos de respuesta garantizados

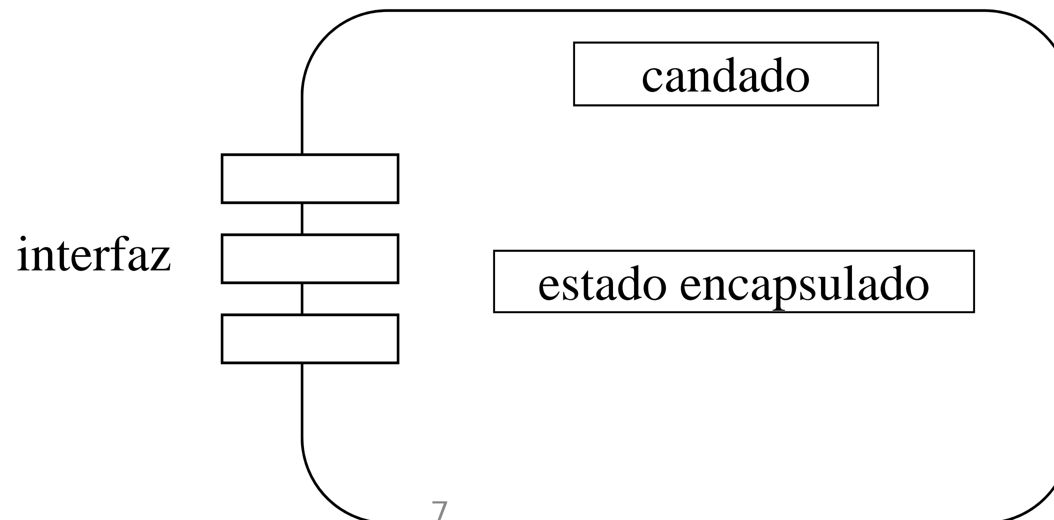
Modelos de concurrencia I

- Procesos versus Threads

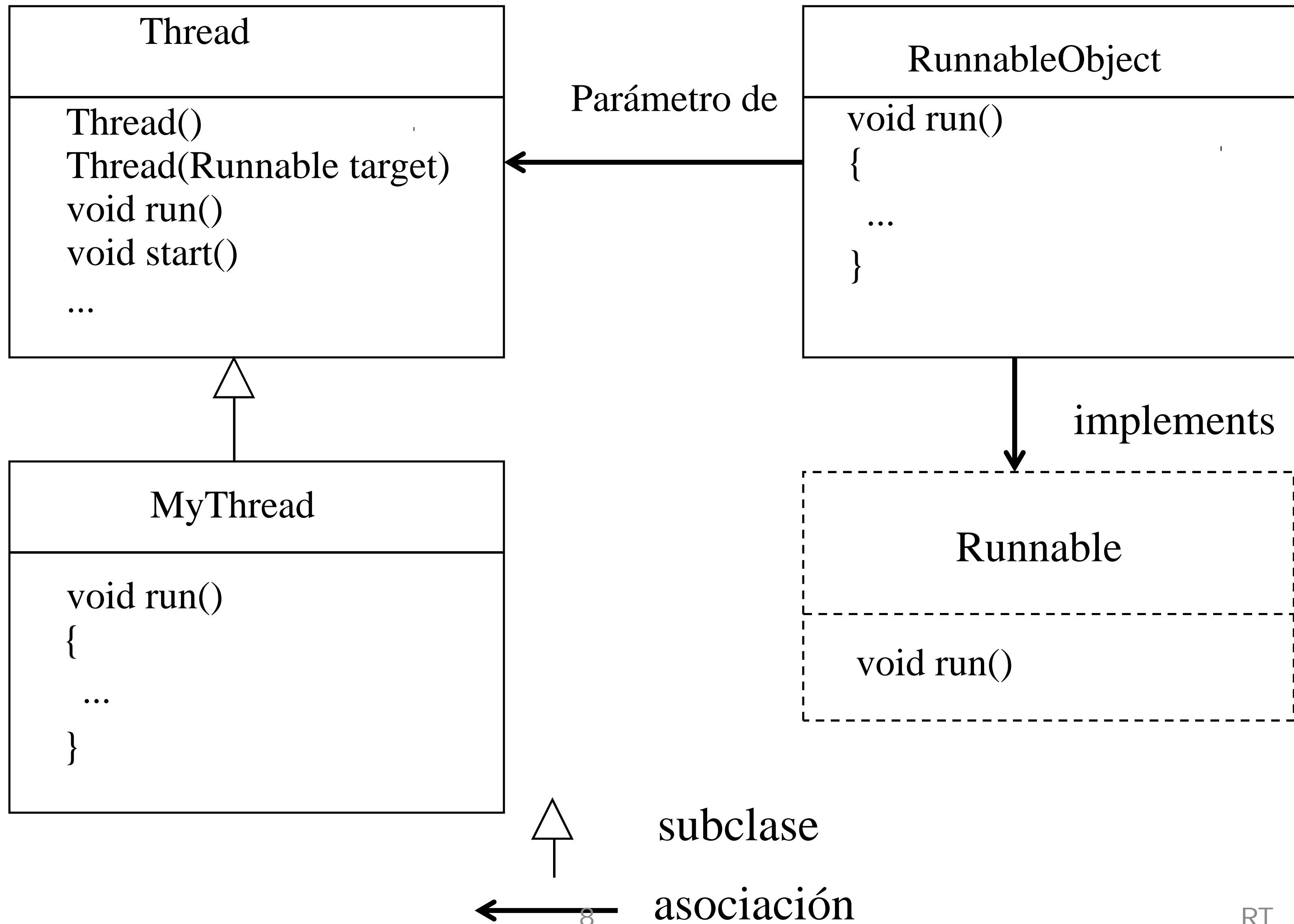


Modelo de concurrencia de Java

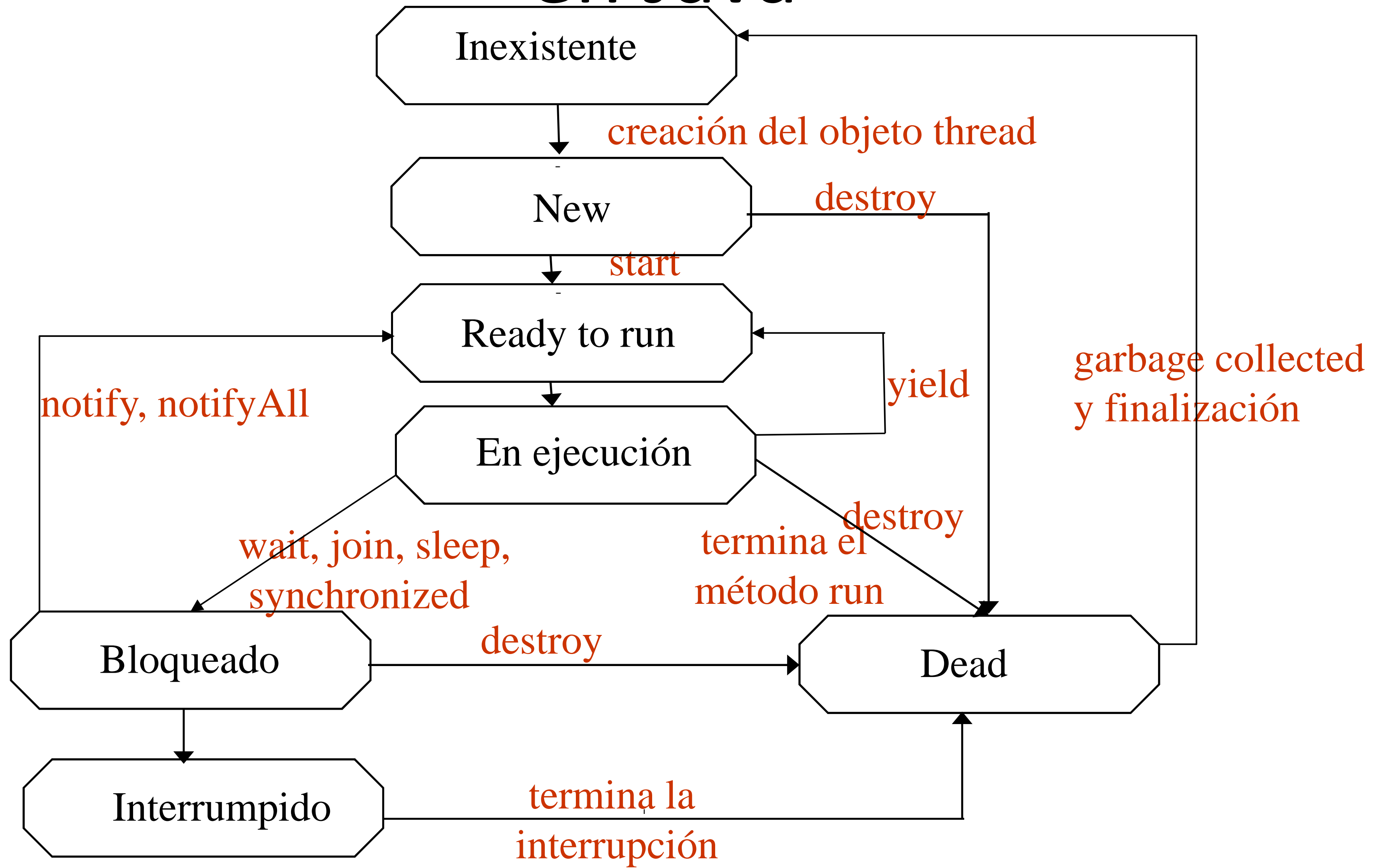
- Comunicación y sincronización
  - Independientemente de cómo se ejecuten las actividades concurrentes, ellas necesitan comunicarse y sincronizar sus ejecuciones
  - Un modelo popular es el de "monitor"
  - Un monitor encapsula un recurso compartido y proporciona una interfaz hacia el recurso en un entorno de exclusión mutua



Threads en Java



Resumen I: Estados de un thread en Java



Sincronización

- Cuando el método `start` ha sido llamado, el thread es elegible para ejecución por el calendarizador del SO
- Si el thread llama su método `wait`, o llama al método `join` en otro objeto, el thread permanece bloqueado y no es elegible para ejecución
- Se vuelve ejecutable como resultado de la llamada al método asociado `notify` por parte de otro thread, o si el thread al que le ha solicitado el join, pasa al estado dead

Métodos synchronized

- Existe un candado de exclusión mutua asociado con cada objeto. Éste no puede ser accesado directamente pero es afectado por:
  - El modificador de métodos synchronized
  - La sincronización de un bloque de instrucciones
- Cuando un método es etiquetado como synchronized, sólo un thread puede ejecutarlo al mismo tiempo. Si otro thread intenta acceder al método utilizado o a otro método synchronized del mismo objeto, el thread será suspendido hasta que el primer thread termine la ejecución del método
- Los métodos no sincronizados no requieren un candado y por lo tanto pueden ser llamados en cualquier momento

Esperando y notificando

- Para obtener la sincronización se requiere de los métodos establecidos en la clase Object

```
public class Object {
  ...

  public final void notify();
  public final void notifyAll();
  public final void wait()
                    throws InterruptedException;
  public final void wait(long millis)
          throws InterruptedException;
  public final void wait(long millis, int nanos)
          throws InterruptedException;
  ...
}
```

Esperando y notificando

- Estos métodos deben ser usados solamente a partir de métodos que han obtenido el candado de un objeto
- Si se llaman sin el candado se arroja la excepción **`IllegalMonitorStateException`**
- El método **`wait`** bloquea al thread que lo invoca y libera el candado asociado con el objeto sobre el que opera
- Para despertar a todos los threads suspendidos se requiere el uso del método **`notifyAll`**

Interrupción de un thread

- Un thread suspendido es puesto en ejecución si es interrumpido por otro thread
- En este caso la interrupción **`InterruptedException`** es lanzada

Bounded Buffer

```
public class BoundedBuffer  {
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  };
```

Bounded Buffer

```
public synchronized void put(int item)
        throws InterruptedException {
    while (numberInBuffer == size) wait();
    last = (last + 1) % size ; // % is modulus
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
};

public synchronized int get()
        throws InterruptedException {
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size ; // % is modulus
    numberInBuffer--;
    notifyAll();
    return buffer[first];
};
}
```

Bounded Buffer – Con Lock

```
import java.util.concurrent.*;
public class BoundedBuffer {

  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer;
  private int size;
  private Lock lock = new ReentrantLock();
  private final Condition notFull =
                 lock.newCondition();
  private final Condition notEmpty =
                 Lock.newCondition();
```

Bounded Buffer – Con Lock

```
public BoundedBuffer(int length) {
  size = length;
  buffer = new int[size];
  last = 0;
  first = 0;
  numberInBuffer = 0;
}
```

Bounded Buffer – Con Lock

```
public void put(int item)
         throws InterruptedException {
  lock.lock();
  try {
    while (numberInBuffer == size)
              notFull.await();
    last = (last + 1) % size;
    numberInBuffer++;
    buffer[last] = item;
    notEmpty.signal();
  } finally {
       lock.unlock();
    }
}
```

Bounded Buffer – Con Lock

```
public int get(int item)
         throws InterruptedException {
  lock.lock();
  try {
    while (numberInBuffer == 0)
              notEmpty.await();
    first = (first + 1) % size;
    numberInBuffer--;
    notFull.signal();
    return buffer[first];
  } finally {
       lock.unlock();
    }
}
```

Prioridades de los threads

```
package java.lang;
public class Thread extends Object
             implements Runnable {
  // constantes
  public static final int MAX_PRIORITY  = 10;
  public static final int MIN_PRIORITY  = 1;
  public static final int NORM_PRIORITY = 5;
  // métodos
  public final int getPriority();
  public final void setPriority(int newPriority);
  public static void yield();
  ...
}
```

Precaución

- Desde la perspectiva real-time, la calendarización de Java y el modelo de prioridades son débiles, en particular:
  - No hay garantía de que el thread ejecutable de mayor prioridad esté siempre ejecutándose
  - Threads de prioridad equivalente pueden o no tener el mismo tiempo de procesador
  - Diferentes prioridades de Java pueden ser traducidas al mismo nivel de prioridad del sistema operativo

Retrasando threads: Clocks

- Java suporta la noción de reloj de pared
- `java.lang.System.currentTimeMillis` regresa el número de milisegundos desde el 1/1/1970 GMT y es utilizado por `java.util.Date` (ver también `java.util.Calendar`)
- Sin embargo, un thread puede ser retrasado solamente mediante la llamada al método `sleep` de la clase `Thread`
- `sleep` provee un retraso relativo (sleep a partir de ahora por X milisegundos y Y nanosegundos)

Tiempo de espera

```
public class TimeoutException extends Exception
{};

public class TimedWait
{
  public static void wait(Object lock, long millis)
      throws InterruptedException, TimeoutException
  {
    // asume que quien llama posee el candado
    long start = System.currentTimeMillis();
    lock.wait(millis);
    if(System.currentTimeMillis() >= start + millis)
        throw new TimeoutException();
  }
}
```

¿Cuál es el problema aquí?

Grupos de threads I

- Un grupo de threads permite que colecciones de threads sean agrupadas y manipuladas juntas en lugar de los individuos
- Cada thread en Java es un miembro de un grupo de threads
- Existe un grupo default asociado con el programa principal y, a menos que se especifique otra cosa, todos los threads creados son puestos en este grupo

Fortalezas del modelo de concurrencia de Java

- La principal fortaleza es que es simple y es soportado directamente por el lenguaje
- Esto permite que muchos de los errores que pueden ocurrir potencialmente usando una interfaz del SO para concurrencia no existan en Java
- La sintaxis del lenguaje y el fuerte chequeo de tipos otorgan cierta protección
- P.ej., no es posible olvidar cerrar un bloque synchronized
- La portabilidad de los programas se refuerza ya que el modelo de concurrencia que usa el programador es el mismo independientemente del SO que el programa finalmente ejecuta

Debilidades

- Falta de soporte para variables de condición
- Pobre soporte para el tiempo absoluto
- No se da preferencia a los threads que continúan su ejecución después de un notify sobre los threads que van a ejecutarse por primera vez
- Pobre soporte a las prioridades