

Programmation concurrente

Notions fondamentales - Threads

Programmation concurrente en java

Threads

□ Rappel : un thread est un fil d'exécution - un travailleur (pointeur de programme, pile, registres,...)

□ Lancement d'un thread en Java :

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



Ce thread ne fait rien et se termine immédiatement.
Il faut lui préciser la tâche à exécuter



Programmation concurrente en java

□ Tâche exécutée par un thread

- Tout le travail se situe dans la méthode `run()` de la classe `Thread`
- La classe `Thread` implémente l'interface *Runnable* qui ne définit qu'une seule méthode :

```
public interface Runnable {  
    public abstract void run();  
}
```

- Pour préciser à un thread la tâche à exécuter. Il y a deux solutions :
 - Sous-classer la classe `Thread` et redéfinir (*Override*) la méthode `run()`
 - Passer un objet de type *Runnable* au constructeur de la classe `Thread`



Programmation concurrente en java

❑ Sous-classe Thread

```
public class MonThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("Do not disturb");  
    }  
}  
  
public MonThread() {  
    super();  
}  
  
public static void main(String[] args) {  
    MonThread t = new MonThread();  
    t.start();  
}
```



Programmation concurrente en java

❑ Implémenter Runnable

```
public class MyRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("Do not disturb");  
    }  
}  
  
public MyRunnable() {  
    super();  
}  
  
public static void main(String[] args) {  
    MyRunnable r = new MyRunnable();  
    Thread t = new Thread(r);  
    t.start();  
}
```



Programmation concurrente en java

❑ Implémenter Runnable

- ❑ A priori plus complexe (2 classes impliquées)
- ❑ Meilleure séparation des rôles
 - ❑ Le Thread : "l'ouvrier"
 - ❑ Le Runnable : "le travail" à faire
- ❑ Un même travail peut-être confié à plusieurs threads



Programmation concurrente en java

- Un même "travail" peut-être confié à plusieurs threads

```
public class TestThreads {  
    public static void main (String [] args){  
        MyRunnable r = new MyRunnable();  
        Thread foo = new Thread(r);  
        Thread bar = new Thread(r);  
        Thread bat = new Thread(r);  
        foo.start();  
        bar.start();  
        bat.start();  
    }  
}
```



Programmation concurrente en java

❑ Implémenter Runnable avec une classe anonyme

```
public class AnonymousRunnable {  
  
    public static void main(String[] args) {  
        Thread t = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Do not disturb");  
            }  
        });  
  
        t.start();  
    }  
}
```



Programmation concurrente en java

- ❑ Sous-classer Thread avec une classe anonyme



Programmation concurrente en java

❑ Constructeurs et attributs de la classe Thread

❑ Principaux constructeurs

- ❑ `Thread()`

- ❑ `Thread(Runnable target)`

- ❑ `Thread(Runnable target, String name)`

- ❑ `Thread(String name)`

❑ Les threads peuvent être nommés

❑ Pour accéder au thread dans lequel le code est exécuté il faut utiliser la méthode statique :

`Thread.currentThread()`



Programmation concurrente en java

□ Main Thread

- Lors de l'exécution d'un programme java (java MonProgramme) la JVM crée un thread dont la méthode run() invoque la méthode main de la classe MonProgramme
- Le programme :

```
public class MonProgramme {  
    public static void main(String[] args) {  
        System.out.println("Le thread est " +  
            Thread.currentThread().getName());  
    }  
}
```

affiche : Le thread est main



Programmation concurrente

Accès concurrents

Programmation concurrente en java

□ Partage de ressources - exclusion mutuelle

- Java définit le mot clé **synchronized** pour démarquer une section de code qui ne doit être exécutée que par un seul thread à la fois
- Une section de code marquée par **synchronized** se traduit par l'acquisition d'un verrou (**monitor lock**) sur un objet
- Tout objet de la classe `java.lang.Object` peut servir de verrou
- Le mot clé **synchronized** peut s'appliquer à une méthode statique, à une méthode d'instance ou à un bloc de code.



Programmation concurrente en java

□ Partage de ressources - exclusion mutuelle

```
class SyncTest {  
    public void doStuff() {  
        System.out.println("non synchronisé");  
        synchronized(this) {  
            System.out.println("synchronisé");  
        }  
    }  
}
```

Synchronisation sur l'objet courant



Programmation concurrente en java

□ Partage de ressources - exclusion mutuelle

```
class SyncTest {  
    private Object lock;  
    public void doStuff() {  
        System.out.println("non synchronisé");  
        synchronized(lock) {  
            System.out.println("synchronisé");  
        }  
    }  
}
```

Synchronisation sur l'objet lock



Programmation concurrente en java

□ Partage de ressources - exclusion mutuelle

```
class SyncTest {  
    public synchronized void doStuff() {  
        System.out.println("synchronized");  
    }  
}
```

est équivalent à :

```
class SyncTest {  
    public void doStuff() {  
        synchronized(this) {  
            System.out.println("synchronized");  
        }  
    }  
}
```



Programmation concurrente en java

□ Partage de ressources - exclusion mutuelle

□ Utilisation du mot clé synchronized sur une méthode statique

□ L'objet qui sert de verrou est l'object de type java.lang.Class qui est associé à la classe dans laquelle la méthode est définie

```
public static synchronized int getCount() {  
    return count;  
}
```

est équivalent à :

```
public static int getCount() {  
    synchronized(MaClasse.class) {  
        return count;  
    }  
}
```



Programmation concurrente en java

❑ Quelle sera la sortie de ce programme ?

```
public class PingPong {  
  
    public static synchronized void main(String[] a) {  
        Thread t = new Thread() {  
            public void run() { pong(); }  
        };  
  
        t.run();  
        System.out.print("Ping");  
    }  
  
    static synchronized void pong() {  
        System.out.print("Pong");  
    }  
}
```



Programmation concurrente en java

❑ Quelle sera la sortie de ce programme ?

```
public class PingPong {  
  
    public static synchronized void main(String[] a) {  
        Thread t = new Thread() {  
            public void run() { pong(); }  
        };  
        t.run();  
        System.out.print("Ping");  
    }  
  
    static synchronized void pong() {  
        System.out.print("Pong");  
    }  
}
```



Piège classique !
Il faut appeler
`t.start();`



Programmation concurrente en java

❑ Interblocage - exemple complexe

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          EuroAmount amount)
    throws InsufficientFundsException{
    synchronized (fromAccount){
        synchronized (toAccount){
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else{
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```

Thread A : transferMoney(myAccount, yourAccount, 10)

Thread B : transferMoney(yourAccount, myAccount, 20)

⇒ Problème

Programmation concurrente en java

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct,
                          final DollarAmount amount)
    throws InsufficientFundsException {

//Helper class
    class Helper {
        public void transfer() throws InsufficientFundsException
        {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }

    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    ...
}
```

```
...
if (fromHash < toHash) {
    synchronized (fromAcct) {
        synchronized (toAcct) {
            new Helper().transfer();
        }
    }
} else if (fromHash > toHash) {
    synchronized (toAcct) {
        synchronized (fromAcct) {
            new Helper().transfer();
        }
    }
} else {
    synchronized (tieLock) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    }
}
}
```

Programmation concurrente

□ Utilisation explicite de verrous

□ En Java

```
Lock lock = new ReentrantLock();  
lock.lock();  
try{ //Critical section }  
finally{  
    lock.unlock();  
}
```


Programmation concurrente

□ Utilisation explicite de verrous

- Intérêt par rapport à `synchronized` → évite des attaques (dédi de service)

```
public class SomeObject {  
  
    // Locks on the object's monitor  
    public synchronized void changeValue() {  
        // ...  
    }  
}  
  
class Malicious{  
    void someMethod() throws InterruptedException{  
        // Untrusted code  
        SomeObject someObject = new SomeObject();  
        synchronized (someObject) {  
            while (true) {  
                // Indefinitely delay someObject  
                Thread.sleep(Integer.MAX_VALUE);  
            }  
        }  
    }  
}
```

→ Solution : ajouter une variable d'instance privée et se synchroniser dessus ou avoir recours à un objet de type `Lock`

Programmation concurrente en java

Utilisation explicite de verrous

Intérêt par rapport à synchronized

- Permet de ne prendre le verrou que si celui-ci est disponible

- `tryLock()`

- Permet de tenter de prendre un verrou avec un timeout

- `tryLock(long, TimeUnit)`

- C'est la meilleure solution pour éviter les Deadlocks

<<Java Interface>> Lock	
lock():void	
lockInterruptibly():void	
tryLock():boolean	
tryLock(long,TimeUnit):boolean	
unlock():void	
newCondition():Condition	

Programmation concurrente

Thread safety

Programmation concurrente

❑ Thread safety

- ❑ Une classe est dite *thread-safe* si elle peut-être utilisée “simultanément” par plusieurs threads
- ❑ Une classe qui ne possède pas d'état est thread-safe

```
public class StatelessFactorizer implements Servlet {  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        encodeIntoResponse(resp, factors);  
    }  
}
```

Programmation concurrente en java

❑ Thread safety - atomicité

- ❑ L'ajout d'un compteur dont la mise à jour est une séquence read-modify-write qui a besoin d'être effectuée de façon atomique casse la propriété de thread-safety

//Not Thread-Safe

```
public class UnsafeCountingFactorizer implements Servlet {  
    private long count = 0;  
  
    public long getCount() { return count; }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        ++count;  
        encodeIntoResponse(resp, factors);  
    }  
}
```



Programmation concurrente en java

❑ Thread safety - atomicité

❑ Solution en utilisant une classe du package `java.util.concurrent.atomic`

```
public class CountingFactorizer implements Servlet {  
    private final AtomicLong count = new AtomicLong(0);  
  
    public long getCount() { return count.get(); }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        count.incrementAndGet();  
        encodeIntoResponse(resp, factors);  
    }  
}
```

Programmation concurrente

□ Atomicité - Exemples d'opérations disponibles

JAVA	AtomicBoolean	<code>public final boolean compareAndSet(boolean expect, boolean update)</code> <code>public final boolean get()</code> <code>public final void set(boolean newValue)</code>
	AtomicInteger	<code>public final int get()</code> <code>public final void set(int newValue)</code> <code>public final boolean compareAndSet(int expect, int update)</code> <code>public final int getAndIncrement()</code> <code>public final int getAndDecrement()</code> <code>public final int getAndAdd(int delta)</code>
Cocoa	Fonctions C	<code>OSAtomicAdd32, OSAtomicAdd64</code> <code>OSAtomicIncrement32, OSAtomicIncrement64</code> <code>OSAtomicDecrement32</code> <code>OSAtomicCompareAndSwap32, OSAtomicCompareAndSwapPtr,</code> <code>OSAtomicTestAndSet</code> <code>OSAtomicTestAndClear</code> <code>OSAtomicOr32</code> <code>OSAtomicAnd32</code> <code>OSAtomicXor32</code>

Programmation concurrente en java

❑ Thread safety

- ❑ Première tentative de mise en cache du dernier nombre et des facteurs associés

```
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get() );
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```



Programmation concurrente en java

❑ Thread safety

- ❑ L'utilisation des variables "atomic" ne règle pas le problème : il faut maintenir la cohérence des 2 variables

```
public class UnsafeCachingFactorizer implements Servlet {  
    private final AtomicReference<BigInteger> lastNumber = new AtomicReference<BigInteger>();  
    private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<BigInteger[]>();  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        if (i.equals(lastNumber.get()))  
            encodeIntoResponse(resp, lastFactors.get() );  
        else {  
            BigInteger[] factors = factor(i);  
            lastNumber.set(i);  
            lastFactors.set(factors);  
            encodeIntoResponse(resp, factors);  
        }  
    }  
}
```



Programmation concurrente en java

❑ Thread safety

- ❑ Mauvaise solution permettant de rendre la classe thread-safe (problème de performance)

```
public class SynchronizedFactorizer implements Servlet {  
    private BigInteger lastNumber;  
    private BigInteger[] lastFactors;  
  
    public synchronized void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        if (i.equals(lastNumber))  
            encodeIntoResponse(resp, lastFactors);  
        else {  
            BigInteger[] factors = factor(i);  
            lastNumber = i;  
            lastFactors = factors;  
            encodeIntoResponse(resp, factors);  
        }  
    }  
}
```



Programmation concurrente en java

□ Thread safety

- Amélioration de la concurrence de la solution permettant de rendre la classe thread-safe

```
public class CachedFactorizer implements Servlet {
    private BigInteger lastNumber;
    private BigInteger[] lastFactors;

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            if (i.equals(lastNumber)) {
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

Programmation concurrente en java

❑ Thread safety

- ❑ Pour régler les problèmes, la meilleure solution est souvent de ne pas partager les objets ou de partager des objets immuables
- ❑ Solutions pour ne pas partager :
 - ❑ Gestion par les développeurs : s'assurer que tous les accès se font depuis le bon thread
 - ❑ Objets confinés à la pile (locale à chaque thread) en utilisant des variables locales
 - ❑ Utilisation de la classe `ThreadLocal` (maintient des copies distinctes pour chaque thread)
- ❑ Objets immuables
 - ❑ Pas de modification de l'état après la construction
 - ❑ Tous les champs sont *final*

Programmation concurrente

Visibilité

Programmation concurrente



☐ Visibilité

- ☐ Notion subtile : contraire à la logique
- ☐ Quand l'écriture et la lecture dans une variable se passent sur des threads différents, il n'y a pas de garantie que le thread lecteur voie une valeur qui a été écrite précédemment par un autre thread.
- ☐ Il peut y avoir un retard avant que la dernière écriture soit visible au thread lecteur
- ☐ Il se peut que le thread lecteur ne voie jamais la valeur écrite

Programmation concurrente

☐ Visibilité

☐ Exemple de problème (en Java)

```
public class NoVisibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready)  
                Thread.yield();  
            System.out.println(number);  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```

Partage sans synchronisation :

ReaderThread peut ne jamais voir les deux écritures dans les variables partagées ou les voir dans l'ordre inverse

Programmation concurrente

☐ Visibilité

- ☐ Raisons de ce qui pourrait passer pour un bug
 - ☐ La levée des contraintes de visibilité et d'ordre permet aux compilateurs et aux JVM de procéder à des optimisations tirant le meilleur parti des fonctionnalités des processeurs modernes (pipeline, multiprocesseurs)
 - ☐ Pour maintenir le pipeline plein, les compilateurs réarrangent l'ordre des opérations
 - ☐ Afin d'accélérer les temps de traitement, les processeurs conservent les valeurs en cache (voire en cache lié à un processeur) ou dans des registres

Programmation concurrente

☐ Visibilité

☐ Quelles sont les sorties possibles de ce programme ?

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start();
        other.start();
        one.join();
        other.join();
        System.out.println("(" + x + "," + y + ")");
    }
}
```

Programmation concurrente

☐ Visibilité

☐ Solutions :

☐ Recourir aux outils de synchronisation(synchronized, locks, ...)

```
public class MutableInteger {  
    private int value;  
  
    public int get() {  
        return value;  
    }  
  
    public void set(int value) {  
        this.value = value;  
    }  
}
```













```
public class SynchronizedInteger {  
    private int value;  
  
    public synchronized int get() {  
        return value;  
    }  
  
    public synchronized void set(int value) {  
        this.value = value;  
    }  
}
```





















Pour plus de détails : <http://download.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility>

Programmation concurrente

☐ Visibilité

- ☐ Autre solution : Utiliser les classes du package `java.util.concurrent.atomic`

<<Java Class>>  AtomicBoolean	
	<code>AtomicBoolean(boolean)</code>
	<code>AtomicBoolean()</code>
	<code>get():boolean</code>
	<code>compareAndSet(boolean,boolean):boolean</code>
	<code>weakCompareAndSet(boolean,boolean):boolean</code>
	<code>set(boolean):void</code>
	<code>lazySet(boolean):void</code>
	<code>getAndSet(boolean):boolean</code>
	<code>toString():String</code>

<<Java Class>>  AtomicInteger	
	<code>AtomicInteger(int)</code>
	<code>AtomicInteger()</code>
	<code>get():int</code>
	<code>set(int):void</code>
	<code>lazySet(int):void</code>
	<code>getAndSet(int):int</code>
	<code>compareAndSet(int,int):boolean</code>
	<code>weakCompareAndSet(int,int):boolean</code>
	<code>getAndIncrement():int</code>
	<code>getAndDecrement():int</code>
	<code>getAndAdd(int):int</code>
	<code>incrementAndGet():int</code>
	<code>decrementAndGet():int</code>
	<code>addAndGet(int):int</code>
	<code>toString():String</code>
	<code>intValue():int</code>
	<code>longValue():long</code>
	<code>floatValue():float</code>
	<code>doubleValue():double</code>

Programmation concurrente en java

☐ Visibilité

☐ Solutions :

☐ Variables Volatile

- ☐ Permet de signaler au compilateur qu'une variable est partagée
 - ☐ Pas de mise en cache
 - ☐ Pas de réordonnancement des instructions
- ☐ Ne bloque pas les threads en cas d'accès concurrent
- ☐ Garantit la visibilité mais pas l'atomicité
 - ☐ Peut conduire à des raisonnements subtiles
- ☐ Toutes les opérations qui précèdent l'écriture d'une variable volatile dans un thread sont visibles par un autre thread après la lecture de cette même variable (comportement spécifique au Java)

Programmation concurrente

Synchronisation

Programmation concurrente

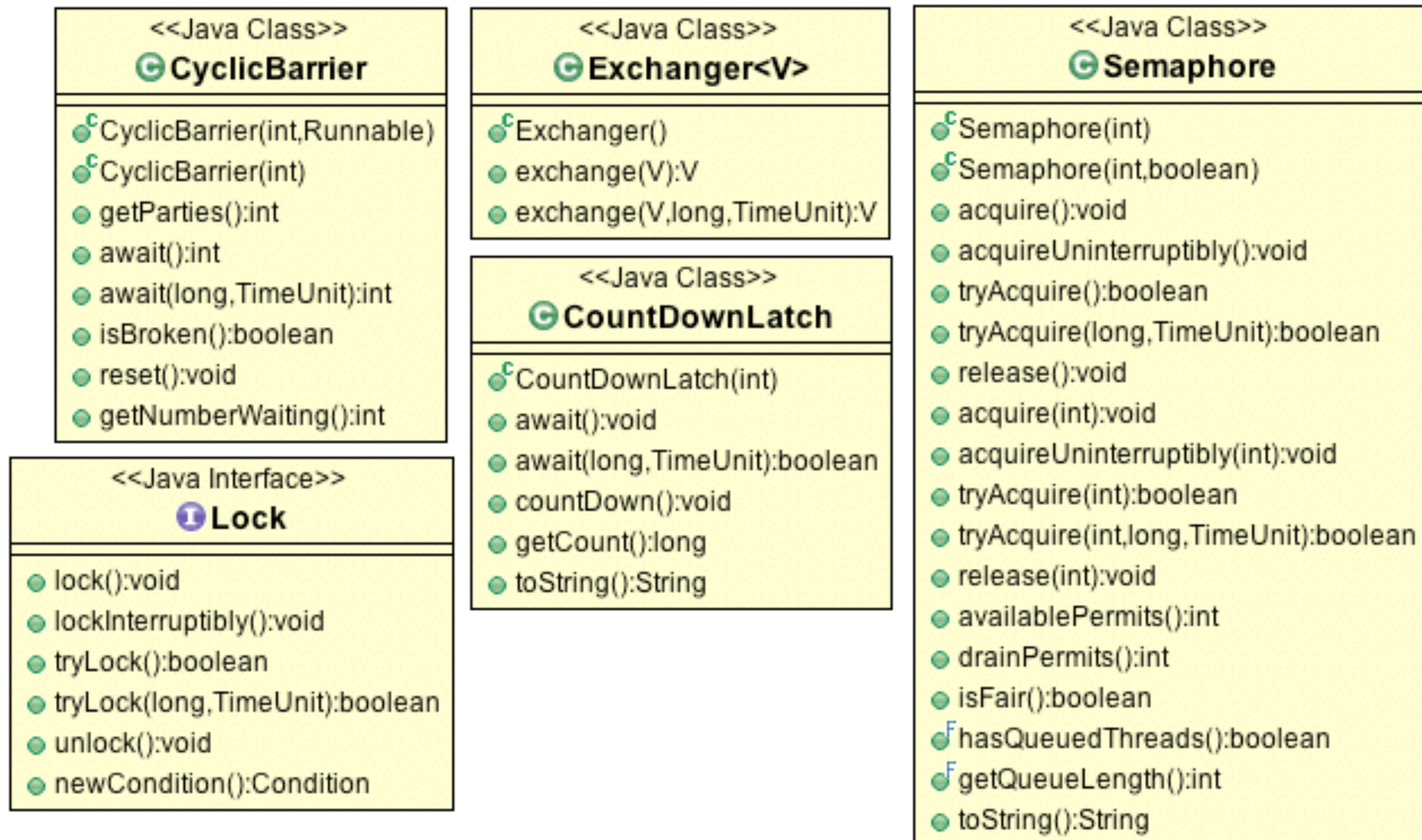


☐ Synchronisation

- ☐ Les packages `java.util.concurrent` et `java.util.concurrent.locks` fournissent des objets pour permettre à des threads de s'attendre mutuellement, d'attendre qu'une condition devienne vraie, de mettre en place des files bloquantes
 - ☐ `CountDownLatch`, `CyclicBarrier`, `Semaphore`, `Exchanger`
 - ☐ `BlockingQueue`
 - ☐ `ReentrantLock`

Programmation concurrente

□ Synchronisation (java)



Programmation concurrente

Du thread aux frameworks de concurrence
Java (android)

Programmation concurrente



☐ Frameworks de concurrence

- ☐ Ecrire un code performant et sans erreur avec des threads est difficile
- ☐ En java et Cocoa, on trouve plusieurs frameworks qui font la partie difficile du travail et qui permettent de découpler le travail à faire du travailleur
- ☐ On soumet des tâches (au sens travail à faire) à un sous-système qui se charge de les exécuter
 - ☐ Le système alloue ou réutilise des threads dont le nombre est optimisé en fonction du nombre de coeurs disponibles
 - ☐ L'utilisateur peut soumettre ou annuler une tâche, attendre la fin et éventuellement le résultat d'une tâche

Programmation concurrente

□ Frameworks de concurrence

□ Contraintes spécifiques à l'environnement mobile

- L'interface utilisateur doit rester réactive ⇨ les tâches longues ne doivent pas s'exécuter dans le thread de gestion de l'interface
 - Communication réseau
 - Calculs importants
 - Requêtes complexes sur une base de données bien remplie
- Si le thread de l'UI est bloqué pendant trop longtemps, l'utilisateur est insatisfait et le système (Android ou iOS) "tue" l'application
 - "Application Not Responding dialog" sur Android
 - Code d'exception "ate bad food" (0x8badf00d) sur iOS

Programmation concurrente



☐ Frameworks de concurrence

- ☐ Les traitements d'arrière plan ne peuvent pas manipuler directement les éléments de l'interface utilisateur
- ☐ Les frameworks d'interface sont tous conçus à partir d'objets qui pour la plupart ne sont pas *thread-safe*
- ☐ Nécessite une façon simple de retourner les informations du traitement d'arrière plan au thread principal

Programmation concurrente

□ Communication UI/Arrière plan sur Android

□ Plusieurs solutions

- Utilisation de la méthode `runOnUiThread(Runnable action)` de la classe `Activity` ou encore `View.post(Runnable action)`
 - S'exécute directement si on est déjà dans le bon thread
 - Ajoute un `Runnable` dans la file des événements à gérer par le thread principal (*UI thread*)
- Utilisation d'un objet `Thread` et d'un objet `Handler`
 - Le thread créé peut envoyer des objets `Runnable` au thread principal ou poster des messages
- Utilisation d'`AsyncTask`
 - Propose un modèle de tâche avec différentes étapes de progression qui s'exécutent sur le thread principal ou en arrière plan



```

public class HandlerDemo extends Activity {
    ProgressBar bar;
    Handler handler=new Handler() {
        @Override
        public void handleMessage(Message msg) {
            bar.incrementProgressBy(5);
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        bar=(ProgressBar)findViewById(R.id.progress);
    }

    public void onStart() {
        super.onStart();
        bar.setProgress(0);

        Thread background=new Thread(new Runnable() {
            public void run() {
                try {
                    for (int i=0;i<20;i++) {
                        Thread.sleep(1000);
                        handler.sendMessage(handler.obtainMessage());
                    }
                } catch (Throwable t) {}
            }
        });

        background.start();
    }
}

```

Exemple d'utilisation
d'un Handler



Exemple d'utilisation
de View.post(...)

```
public void onClick(View v) {  
    String url = "http://example.com/image.png";  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap bitmap = loadImageFromNetwork(url);  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(bitmap);  
                }  
            });  
        }  
    }).start();  
}
```



Programmation concurrente

□ Communication UI/Arrière plan sur Android

□ La classe Handler propose les méthodes suivantes

□ Envoi d'un message

□ `sendMessage(Message)`

□ `sendMessageAtFrontOfQueue(Message)` pour gérer une notion de priorité

□ `sendMessageAtTime(Message,long)` et `sendMessageDelayed(Message,long)` pour gérer l'instant auquel le message sera délivré

□ Envoi d'un bloc de code à exécuter (méthodes également disponibles sur la classe View)

□ `post(Runnable)`

□ `postAtFrontOfQueue(Runnable)`

□ `postAtTime(Runnable,long)`, `postDelayed(Runnable, long)`



Programmation concurrente

❑ Communication UI/Arrière plan sur Android



❑ AsyncTask : 4 étapes

- ❑ **onPreExecute()**, invoquée sur le thread de l'UI immédiatement après le lancement de la tâche par `execute` (préparation de la tâche, par ex. affichage d'une progress bar)
- ❑ **doInBackground(Params...)**, invoquée sur le thread d'arrière plan immédiatement après la fin de `onPreExecute()`. Cette méthode reçoit un tableau de paramètres et doit retourner le résultat du calcul qui sera passé à la dernière étape.
- ❑ Cette tâche peut utiliser **publishProgress(Progress...)** pour communiquer son état d'avancement. Le tableau `Progress` passé en paramètre est transmis à la méthode **onProgressUpdate(Progress...)** qui est invoquée sur le thread de l'UI
- ❑ **onPostExecute(Result)**, invoqué sur le thread de l'UI après la fin de la méthode `doInBackground()`. Le résultat retourné par `doInBackground()` est passé en paramètre.

Programmation concurrente

❑ Communication UI/Arrière plan sur Android

- ❑ Classe générique : `AsyncTask<Params,Progress,Result>`
- ❑ `AsyncTask` : méthode à redéfinir obligatoirement (abstract)
 - ❑ **`Result doInBackground(Params... params)`**
- ❑ `AsyncTask` : méthode que l'on peut redéfinir
 - ❑ **`void onPreExecute()`**
 - ❑ **`void onProgressUpdate(Progress... values)`**
 - ❑ **`void onPostExecute(Result result)`**
 - ❑ **`void onCancelled(Result result)`** (*UI Thread*)
- ❑ Exécution
 - ❑ `final AsyncTask<Params, Progress, Result> execute(Params... params)`



Programmation concurrente

□ Communication UI/Arrière plan sur Android

□ AsyncTask - exemple

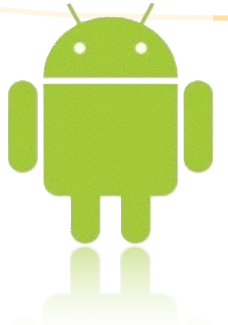
```
public void onClick(View v) {  
    new DownloadImageTask().execute("http://example.com/  
image.png");  
}
```

```
private class DownloadImageTask extends  
AsyncTask<String, Void, Bitmap> {  
    protected Bitmap doInBackground(String... urls) {  
        return loadImageFromNetwork(urls[0]);  
    }  
  
    protected void onPostExecute(Bitmap result) {  
        mImageView.setImageBitmap(result);  
    }  
}
```



Programmation concurrente

❑ Communication UI/Arrière plan sur Android



❑ AsyncTask

- ❑ Il existe principalement deux méthodes permettant de démarrer l'exécution d'une AsyncTask

- ❑ `final AsyncTask<Params, Progress, Result> execute(Params... params)`

- ❑ `final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec, Params... params)`

- ❑ Selon les versions d'Android, la méthode `execute` transmet la tâche à un exécuteur par défaut qui traite les tâches séquentiellement ou de manière concurrente (les dernières versions reviennent à la solution séquentielle). Pour créer plusieurs tâches qui s'exécutent "simultanément" il faut utiliser :

```
myAsyncTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, toto, tutu)
```

Programmation concurrente

❑ Communication UI/Arrière plan sur Android

❑ AsyncTask



```
public final AsyncTask<Params, Progress, Result> execute (Params... params)
```

Added in [API level 3](#)

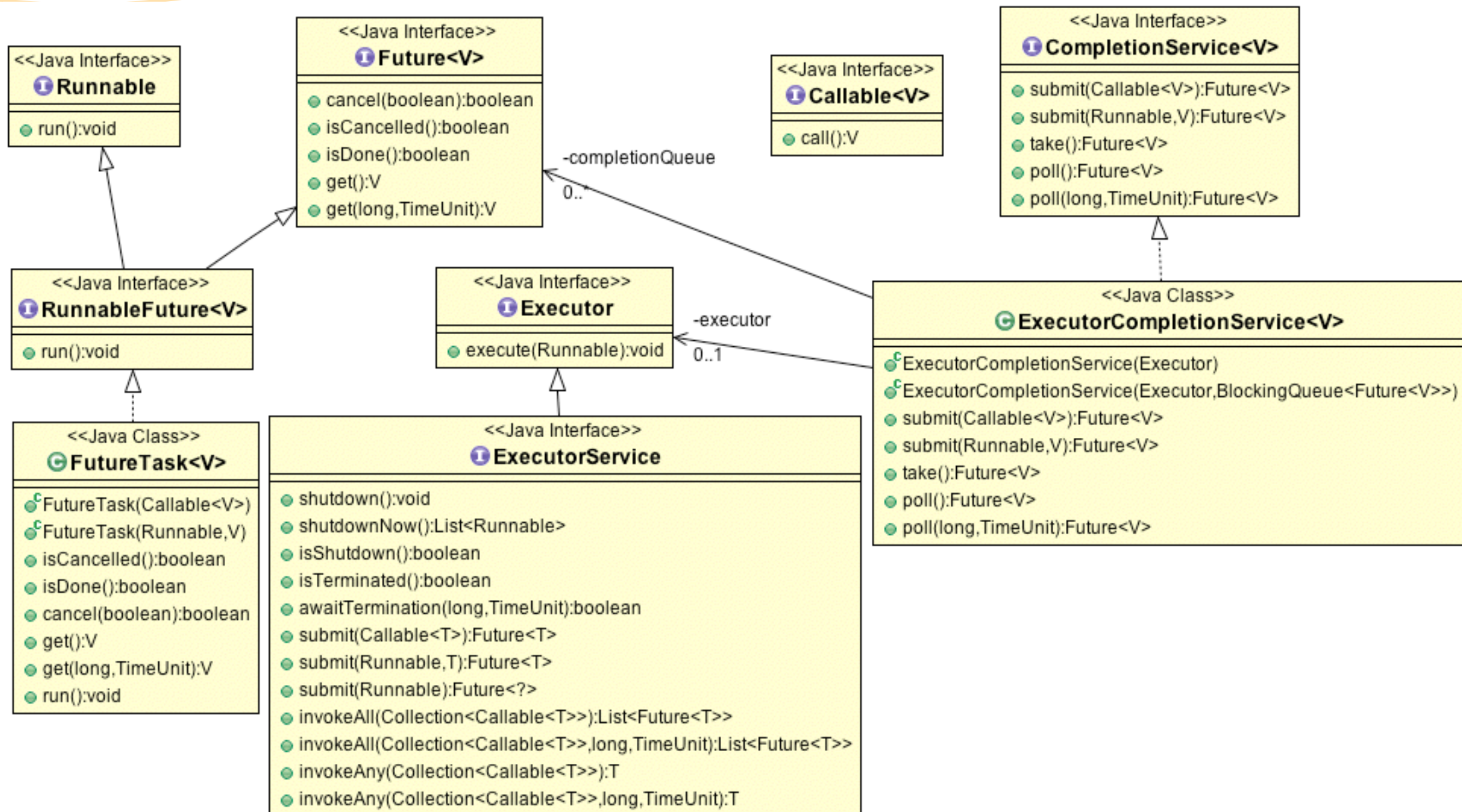
Executes the task with the specified parameters. The task returns itself (this) so that the caller can keep a reference to it. Note: this function schedules the task on a queue for a single background thread or pool of threads depending on the platform version. When first introduced, AsyncTasks were executed serially on a single background thread. Starting with [DONUT](#), this was changed to a pool of threads allowing multiple tasks to operate in parallel. Starting [HONEYCOMB](#), tasks are back to being executed on a single thread to avoid common application errors caused by parallel execution. If you truly want parallel execution, you can use the `executeOnExecutor(Executor, Params...)` version of this method with [THREAD_POOL_EXECUTOR](#); however, see commentary there for warnings on its use. This method must be invoked on the UI thread.

Programmation concurrente

❑ Framework `java.util.concurrent`

- ❑ `AsyncTask` s'appuie sur le concept d'executor qui est au coeur du package `java.util.concurrent`
- ❑ Séparation claire entre les tâches à effectuer (au sens "travail") et les travailleurs
- ❑ Il existe des Executor avec 1 seul thread, un pool fixe de thread ou un pool de thread de taille variable
- ❑ Pour dimensionner les pools de threads, on peut utiliser
`int nCPU = Runtime.getRuntime().availableProcessors()`
- ❑ Pour des tâches avec pas ou peu d'entrées/sorties : optimum `nCPU + 1`
- ❑ Si les tâches recourent à des entrées/sorties il faut augmenter le nombre de threads (il existe des formules mais rien ne remplace un benchmark des performances en situation réaliste)



















Programmation concurrente



Programmation concurrente

□ Framework java.util.concurrent

- Obtenir un executor → fabriques statiques de la classe Executors (ou sur Android, les 2 variables statiques de AsyncTask (THREAD_POOL_EXECUTOR et SINGLE_THREAD_EXECUTOR))

<<Java Class>>	
 Executors	
java.util.concurrent	
	<u>newFixedThreadPool(int):ExecutorService</u>
	<u>newFixedThreadPool(int,ThreadFactory):ExecutorService</u>
	<u>newSingleThreadExecutor():ExecutorService</u>
	<u>newSingleThreadExecutor(ThreadFactory):ExecutorService</u>
	<u>newCachedThreadPool():ExecutorService</u>
	<u>newCachedThreadPool(ThreadFactory):ExecutorService</u>
	<u>newSingleThreadScheduledExecutor():ScheduledExecutorService</u>
	<u>newSingleThreadScheduledExecutor(ThreadFactory):ScheduledExecutorService</u>
	<u>newScheduledThreadPool(int):ScheduledExecutorService</u>
	<u>newScheduledThreadPool(int,ThreadFactory):ScheduledExecutorService</u>
	<u>unconfigurableExecutorService(ExecutorService):ExecutorService</u>
	<u>unconfigurableScheduledExecutorService(ScheduledExecutorService):ScheduledExecutorService</u>
	<u>defaultThreadFactory():ThreadFactory</u>
	<u>privilegedThreadFactory():ThreadFactory</u>
	<u>callable(Runnable,T):Callable<T></u>
	<u>callable(Runnable):Callable<Object></u>
	<u>callable(PrivilegedAction<?>):Callable<Object></u>
	<u>callable(PrivilegedExceptionAction<?>):Callable<Object></u>
	<u>privilegedCallable(Callable<T>):Callable<T></u>
	<u>privilegedCallableUsingClassLoader(Callable<T>):Callable<T></u>

Programmation concurrente

□ Framework java.util.concurrent

- Exemple : rendu d'un contenu web (analyse du fichier html / téléchargement des images et affichage) → séquentiel

```
public abstract class SingleThreadRenderer {  
    void renderPage(CharSequence source) {  
        renderText(source);  
        List<ImageData> imageData = new ArrayList<ImageData>();  
        for (ImageInfo imageInfo : scanForImageInfo(source))  
            imageData.add(imageInfo.downloadImage());  
        for (ImageData data : imageData)  
            renderImage(data);  
    }  
  
    abstract void renderText(CharSequence s);  
    abstract List<ImageInfo> scanForImageInfo(CharSequence s);  
    abstract void renderImage(ImageData i);  
}
```

Programmation concurrente

□ Framework
java.util.concurrent

□ Exemple : rendu d'un contenu web (analyse du fichier html / téléchargement des images et affichage) → version concurrente

```
public abstract class FutureRenderer {
    private final ExecutorService executor = Executors.newCachedThreadPool();

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
            new Callable<List<ImageData>>() {
                public List<ImageData> call() {
                    List<ImageData> result = new ArrayList<ImageData>();
                    for (ImageInfo imageInfo : imageInfos)
                        result.add(imageInfo.downloadImage());
                    return result;
                }
            };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // Re-assert the thread's interrupted status
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Programmation concurrente

□ Framework
java.util.concurrent

□ Exemple : rendu d'un contenu web (analyse du fichier html / téléchargement des images et affichage) → version concurrente

□ Concurrency améliorée

```
public abstract class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) {
        this.executor = executor;
    }

    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });








        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Programmation embarquée et mobile



Bibliographie

-  **Programmation concurrente en Java**, Brian Goetz et al.
-  Effective Java (2nd edition), Joshua Bloch
-  Java(TM) Puzzlers: Traps, Pitfalls, and Corner Cases, Joshua Bloch et Neal Gafter
-  Refactoring: Improving the Design of Existing Code, Martin Fowler
-  Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides
-  Head First Design Patterns, Freeman
-  Cocoa Design Patterns, Erik M. Buck et Donald A. Yacktman