Chapter 12: Concurrency

-**Multitasking**: your OS’s ability to have more than one program working at what seems like the same time.

-**Multithreaded programs** extend the idea of multitasking: Individual programs will appear to do multiple tasks at the same time. Each task is executed in a **thread** (thread of control). Programs that can run more than one thread at once are **multithreaded**.

-**Multiple processes** and **multiple threads**: Each process has a complete set of its own variables, threads share the same data.

-Multithread is extremely useful in practice. Example: browser can simultaneously download images. Web server serves concurrent requests.

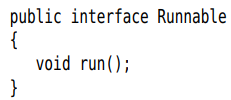
-This chapter shows you how to add multithreading capability to Java app. Concurrent programming can get very complex. We cover all the tools an app programmer is likely to need.

-**Advanced reference**: Java Concurrency in Practice

# 12.1 What are Threads?

-Simple procedure for running a task in a separate thread

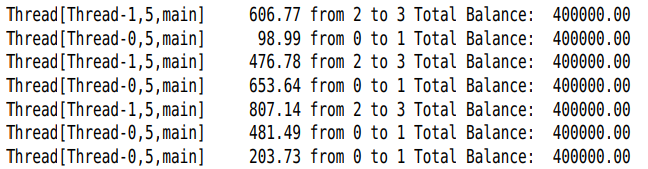
+Place the code for the task in **run()** in **Runnable**

****

+**Construct** a **Thread object**: 

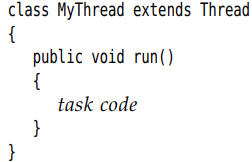
+Start the thread: 

-Output when you run 2 threads:



+The output of 2 threads is interleaved -> **run concurrently**. The **remainder** of this **chapter** controls the **interactions between threads**.

-**.**: You can define a thread (not recommend)



-**Caution**: Don’t call run() of Thread class. It executes the task in the **same thread**.

# 12.2 Thread States

-Thread can be in one of **6 states**: New, Runnable, Blocked, Waiting, Timed waiting, Terminated.

12.2.1 New Threads  
-When you create a thread with **new**

-The thread is not yet running.

12.2.2 Runnable Threads

-When you invoke **start()**

-It may or may not be running.

-Preemptive scheduling systems give each runnable thread a slice of time to perform its task. When this time is exhausted, the OS preempts the thread and gives another thread an opportunity to work. When selecting the next thread, the OS consider the thread **priorities**

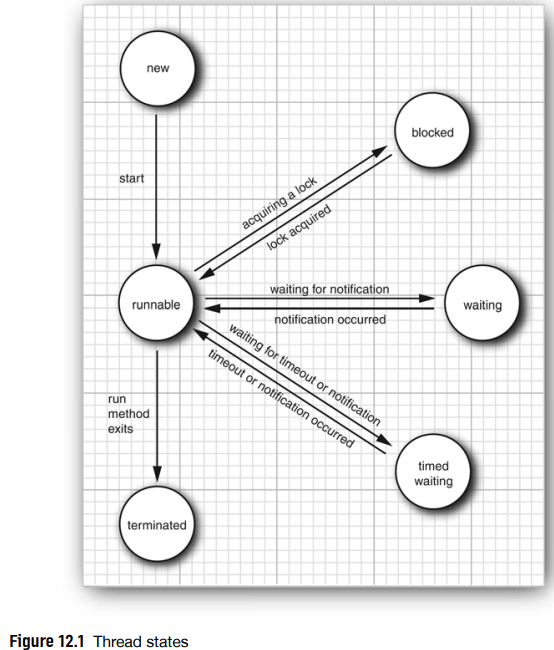
12.2.3 Blocked and Waiting Threads

-It is temporarily inactive: It doesn’t execute any code and consumes minimal resources.

-When the **thread** tries to **acquire** an intrinsic **object** **lock** that is currently **held** by **another thread**, it become **locked**. The **thread** becomes **unblocked** when **all other thread** have **relinquished** the **lock** and the thread scheduler has allowed this thread to hold it.

-When the **thread** **waits** for another thread to notify the scheduler of a condition, it enters the **waiting state**: Object.wait, Thread.join, wait Lock or Condition. In practice, the difference between the blocked and waiting state is not significant

-Some methods have a **timeout** parameter. Calling them causes the thread to enter the **timed waiting state**: Thread.sleep, Object.wait, Thread.join, Lock.tryLock, Condition.await



12.2.4 Terminated Threads

-A thread is **terminated** because of 2 reasons: **natural death** ( run() exits) and **uncaught exception** terminates the run()

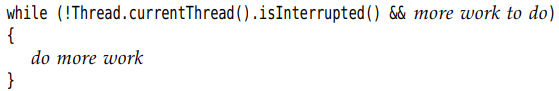
-API: p743 join(), getState, stop, suspend

# 12.3 Thread Properties

12.3.1 Interrupting Threads

-Other than with deprecated stop(), there is no way to force a thread to terminate. But interrupt() can request termination of a thread. When it is called, the **interrupt status** (boolean flag) of the thread is set.

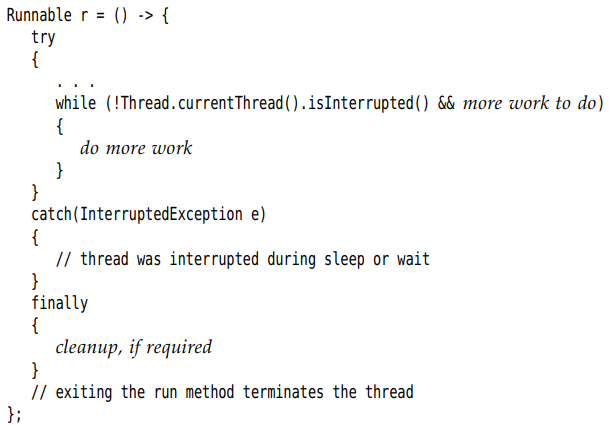
+Find out if the interrupt status was set:



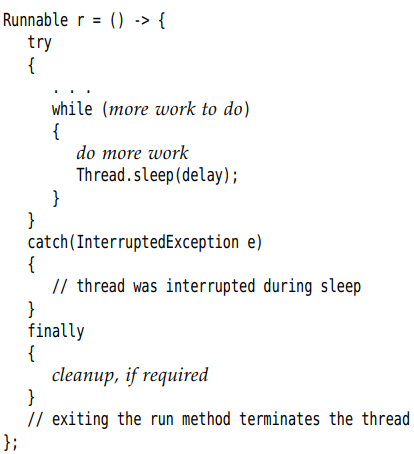
+Block thread cannot check the interrupted status. You can terminate it by an InterruptedException

-The interrupted thread can decide how to react to the interruption.

+Quite commonly, a thread will interpret an interruption as a request for termination:



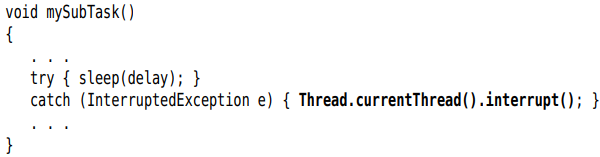
+If you call sleep() when the interrupted status is set, it clear the status and throw exception



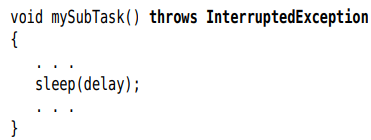
-Note: static interrupted() – check and set the interrupted set to false, isInterrupted() – check the status only.

-Handle InterruptedException:

+Call Thread.currretnThread().interruput()



+Even better, throw and drop the try block:



-API: p746 interrupt(), static interrupted(), isInterrupted(), currentThread()

12.3.2 Daemon Threads

-Turn a thread into a daemon thread: 

-A daemon is a thread that serve others. Example: timer threads send regular timer ticks to other threads, threads that clean up stale cache entries. When only daemon threads remain, the virtual machine exits.

-API: p747 setDaemon(boolean)

12.3.3 Thread Names

-By default, threads have name like Thread-2. You can set



12.3.4 Handlers for Uncaught Exceptions

-**run()** **cannot** throw any **checked exception**, but it can be terminated by an unchecked exception, the thread dies.

-Just before the thread dies, the **exception** is **passed** to **handler** for uncaught exception. The handler must belong to class that implements **Thread.UncaughtExceptionHandler** interface:

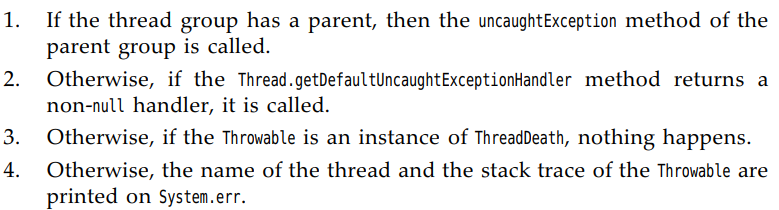


-You can install a handle into any thread: 

+Install a default handler for all threads: 

-If you don’t install a default handler, the default handler is null. If you don’t install a handler for an individual thread, the handler is the thread’s **ThreadGroup** object

-ThreadGroup implements Thread.UncaughtExceptionHandler, its uncaughtException() take the action:



+This is the stack trace you have seen many times.

-API: p748

12.3.5 Thread Priorities

-In Java, every thread has a **priority**. By default, a thread inherits the priority of the thread that constructed it.

-(): change the priority between MIN\_PRIORITY (1) and MAX\_PRIORITY (10) , NORM\_PRIORITY (5)

-When the thread scheduler has a chance to pick a new thread, it prefers thread with higher priority.

-However, thread properties are **highly system-dependent.** The java thread priorities are mapped to the priority levels of the host platform.

->You should not use java thread priorities nowadays.

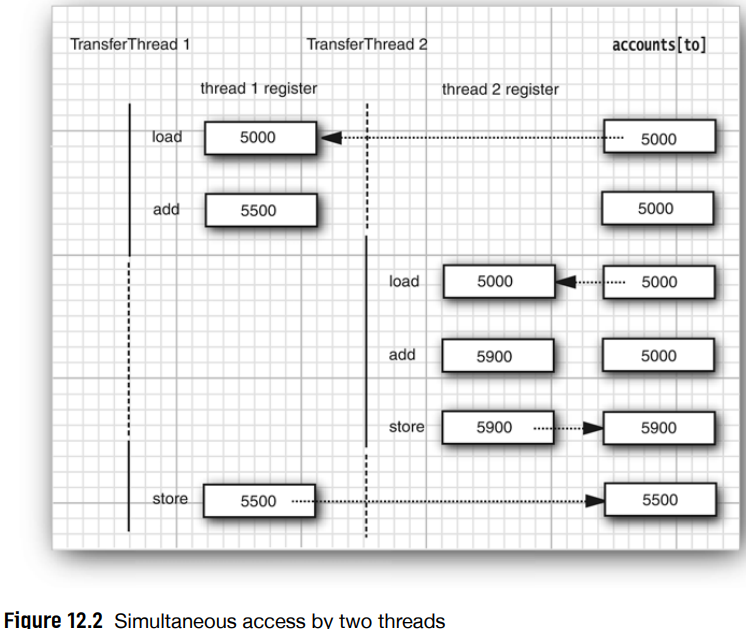
# 12.4 Synchronization

-What happens if **2 threads** have **access** to the **same object** and each calls a method that **modifies** the state. Corrupted object can result. This situation is **race condition**.

12.4.1 An Example of a Race Condition

-To avoid corruption of shared data by multiple threads, you must learn how to **synchronize the access**.

12.4.2 The Race Condition Explained

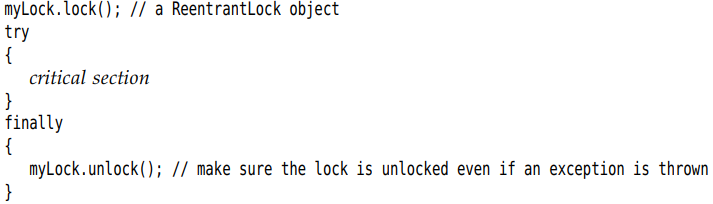


-The real problem is that the work of transfer() can be interrupted in the middle. We need to make the **method** runs to **completion** before the thread loses control.

12.4.3 Lock Objects

-There are 2 mechanisms for **protecting** a code **block** from concurrent access: **synchronized** keyword and **ReentrantLock** class.

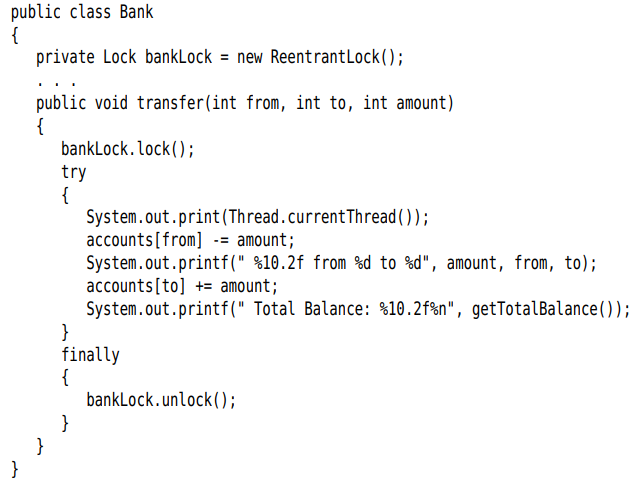
-**synchronized** provides a lock and an associated condition.

-The basic outline for protecting code block with **ReentrantLock**: 

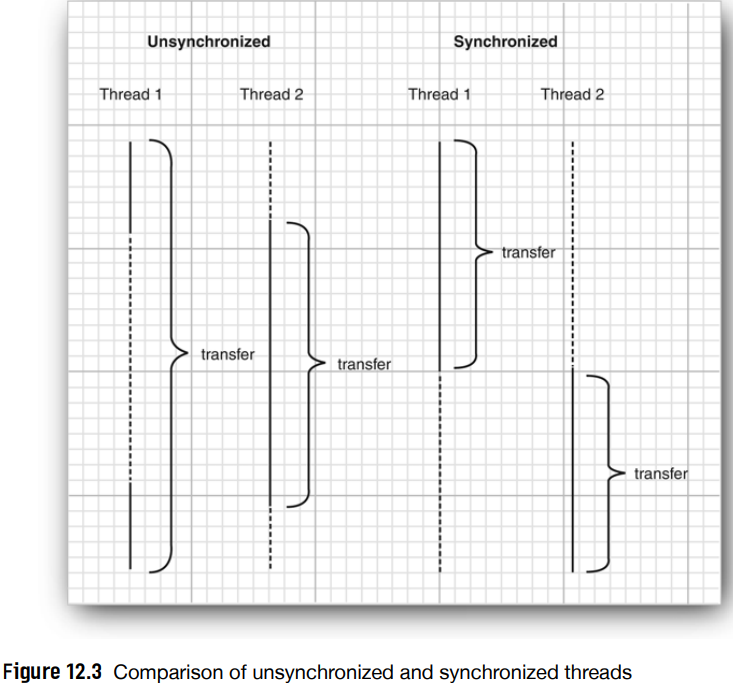
+This construct guarantees that only one thread at a time can enter the critical section. When one thread **locks()** the **lock object**, **no other thread** can get **past** the lock(). When other threads call lock(), they are deactivated until the 1st thread unlocks the lock object.

+**Note**: When using locks cannot use **try-with-resourcces**

-Example: Use a lock to protect transfer()



+Suppose one thread call transfer() and gets **preempted** before it is done. A 2nd thread transfer(), it **cannot** **acquire** the **lock** and is blocked in the call to lock(). It is deactivated and must wait for the 1st thread to finish transfer(). When 1st thread unlock the lock, then 2nd can proceed.



+Each Bank object has its own ReentrantLock object. If 2 threads try to access the same Bank object, the lock serves to serialize the access.

+The lock is called **reentrant**. The lock has a **hold count** that keeps track of the nested calls to lock().

+Code protected by a lock can call another method that uses the same locks.

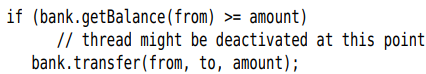
-In general, you will want to protect blocks of code that update or inspect a shared object, so these operations run to completion before another thread can use the same object.

-**Caution**: Be careful to ensure that the code in critical section is not bypassed by throwing an exception.

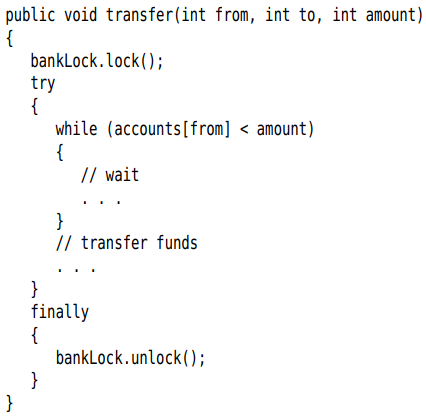
-API: p758

12.4.4 Condition Objects

-Often, a thread enters a critical section only to discover that it can’t proceed until a condition is fulfilled. Use a **condition object** to **manage threads** that have **acquired** a **lock** but cannot do useful work.

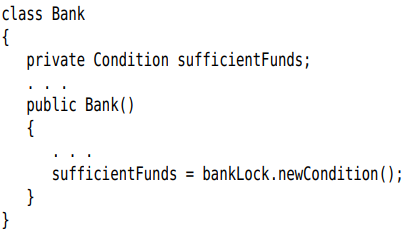


+Protect both the test and the transfer action with lock



+What do we do when there is not enough money in the account. This is where **condition objects** come in.

-A **lock object** can have one or more associated condition objects. Use **newCondiont()** to construct. Give it a name that evokes the condition that it presents:



-If the **condition is not available**:  The current thread is **deactivated** and **give up the lock**. Another thread that can do.

-Thread waiting to acquire a lock and thread called await(): When a thread call **await()**, it enters a **wait set** for that condition. Thread is not made runnable when lock is available, it is **deactivated** until another thread call **signalAll()** on the same condition.

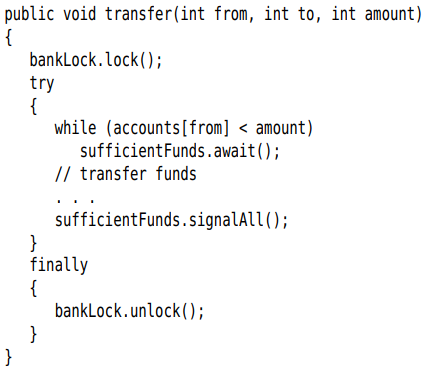
-When another thread has transferred money, it should call . It **reactivates** all **threads waiting** for the condition -> removed from wait set -> are runnable -> acquire the lock and continue where it left off: test the condition again. => **signalAll()** signals to waiting threads that it **may be** fulfilled and it’s worth **checking** for the **condition** again

-**Note**: In general, **await()** should be inside a **loop**:



-Call **signalAll()** whenever the **state** of an **object** **changes** in a way that might be advantageous to waiting thread.

+Example: signalAll() when finishing the funds transfer



+Note that signalAll() only unblocks the waiting threads.

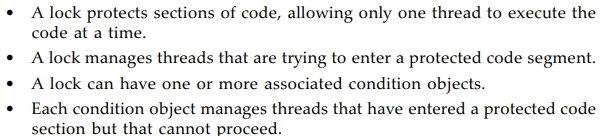
+singal() unblock a single thread from the wait set randomly. If no the chosen thread cannot proceed, it becomes blocked. If no other thread calls signal() again, the system **deadlocks**.

-**Note**: When a thread owns the lock of condition, it can only call await, singalAll() or signal().

-API: p764

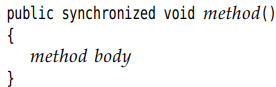
12.4.5 The synchronized Keyword

-Summarize about locks and conditions

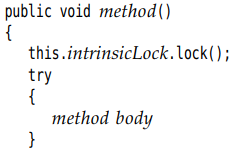


-Lock and Condition interfaces give programmer a high degree of control over locking. But in most situations, you don’t need that control.

-Every object in Java has an **intrinsic lock**. If a method is declared with **synchronized**, the object’s lock **protects** the **entire method**. To call the method, a thread must acquire the object lock.



Is the equivalent of





-The intrinsic object has a single associated **condition**. +**wait()** adds a thread to the wait set

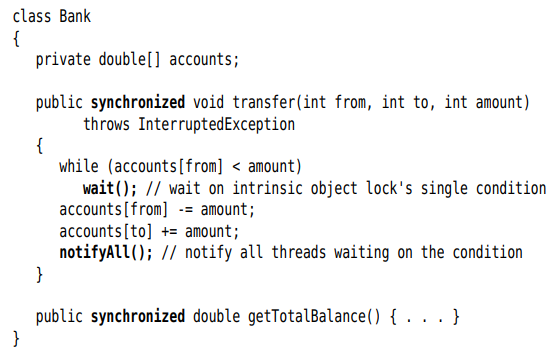
+**notifyAll()/notify()**: unblock waiting threads.

=>wait() and notifyAll() are the equivalent of



+**Note**: wait(), notifyAll(), notify() are final methods of **Object** class. Condition class have await(), signalAll(), signal(). Don’t conflict them.

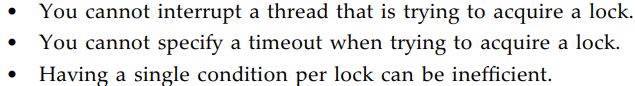
-Bank class:



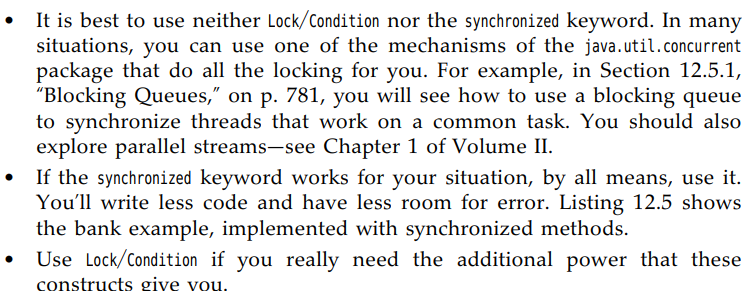
-**Tip**: Synchronized method are **straightforward**. But beginner often struggle with conditions. Before you use wait/notifyAll, consider one of constructors in 12.5

-You can declare **static method** as **synchronized**. It acquires the intrinsic lock of the class object.

-Limitation of intrinsic locks and conditions:



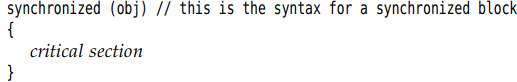
-Recommendation to choose Lock/Condition objects or synchronized methods:



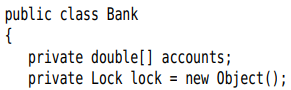
-API: p768 wait(), notify, notifyAll,

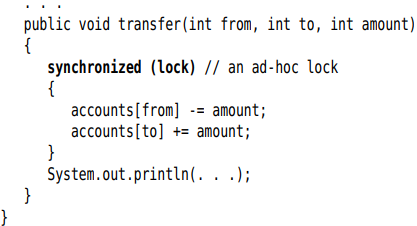
12.4.6 Synchronized Blocks

-A thread can acquire the lock by calling a synchronized method. There is a 2nd mechanism: **synchronized block**

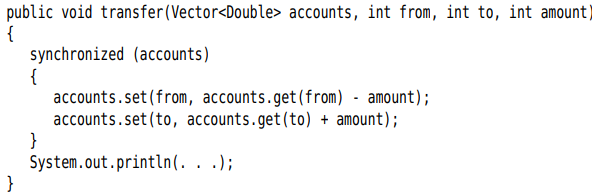


-You can find “ad hoc” locks:



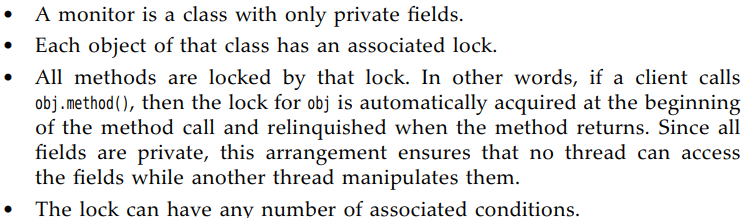


-Sometimes, programmers use the lock of an object to implement additional atomic operations – **client-side locking.**



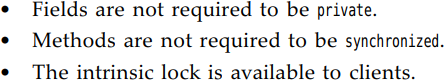
12.4.7 The Monitor Concept

-**Monitor** properties:



-Java designer loosely adapted the monitor concept. Every object in Java has an intrinsic lock and condition. Synchronized method acts like a monitor method. The condition variable is accessed by wait/notify/notifyAll

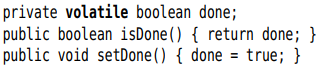
-Java objects differs from a monitor in 3 ways:



12.4.8 Volatile Fields

-“If you write a variable which may next be read by another thread, or you read a variable which may have last been written by another thread, you must use synchronization”– Brian Goetz

-**volatile** keyword offers a **lock-free mechanism** for **synchronizing access** to an **instance field**. The field may be concurrently updated by another thread.



+The compiler will insert the code to ensure that a change to the done variable in one thread is visible from any other thread that reads the variable.

12.4.9 Final Variables

-It is safe to access a shared filed when it is declared final.

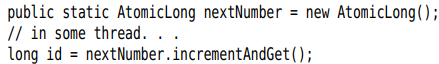


+But the operations on the map are not thread-safe->still need synchrionization.

12.4.10 Atomics

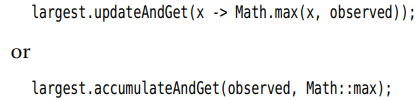
-In Java, **atomicity** refers to operations that are performed as a single, indivisible step, ensuring that they are **thread-safe** and cannot be interrupted.

-java.util.concurrent.atomic package guarantee atomicity operations without using locks.



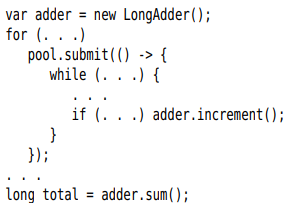
+It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

+There are methods for atomically setting, adding, subtracting values. But if you want to make a more complex update, use **compareAndSet()**



+**getAndUpdate()** and **getAndAccumulate()** return the old value.

-When you have a very large number of threads accessing the same atomic values, use **LongAdder** and **LongAccumulator**.

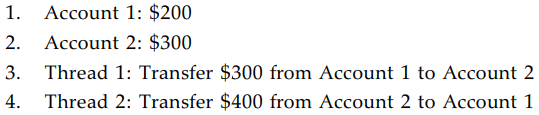
 

+The operation must be associative and commutative.

+There are also **DoubleAdder** and **DoubleAccumulator**

12.4.11 Deadlocks

-**Deadlock**: All threads get **blocked**



-There is nothing in Java to avoid or break deadlocks. You must design the program to ensure deadlock situation cannot occur.

12.4.12 Thread-Local Variables

-Sometimes, you can avoid sharing by giving each thread its own instance by **ThreadLocal** helper class:



+Access the actual formatter:



-Example: Generate random numbers in threads:



+**ThreadLocalRandom.current()** returns an instance of Random class that is unique to the current thread.

-API: p779

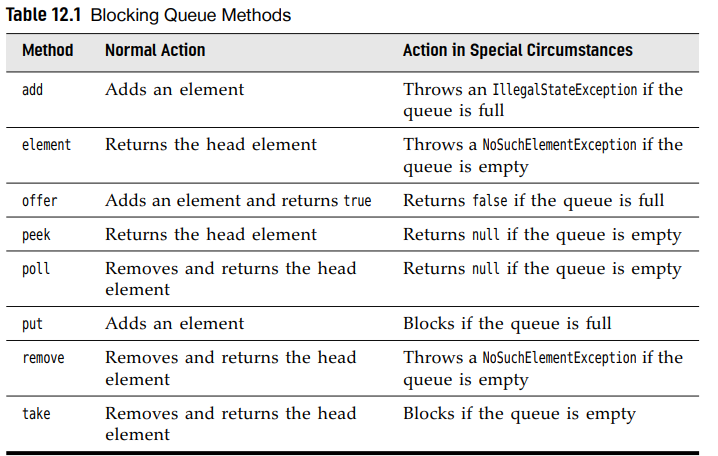
12.4.13 Why the stop and suspend Methods are Deprecated

# 12.5 Thread-Safe Collections

-You can protect a shared data structure by supplying a lock, but it’s easier to choose a thread-safe implementation instead.

12.5.1 Blocking Queue

-Queue lets you safely hand over data from one thread to another:



+offer() and poll() can go with a timeout:

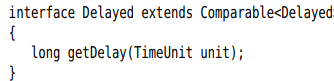
 try for 100 milisec to insert an element to the tail.

-Java.util.concurrent package supplies variations of blocking queues:

+LinkedBlockingQueue, ArrayBlockingQueue

+PriorityBlockingQueue

+DelayQueue contains object that implement Delayed:

 Elements can only be removed from if their delay has elapsed.

-TransferQueue interface: used by LinkedTransferQueue

-APIs: p787

12.5.2 Efficient Maps, Sets, and Queues

-java.util.concurrent supplies efficient implementation for maps, sorted sets and queues: ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, ConcurrentLinkedQueue.

+These collections allow concurrent access to different parts of data structure.

+size() does not necessarily operate in constant time. ConcurrentHashMap have mappingCount() return long.

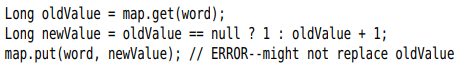
-The collections return **weakly consistent** iterators. The concurrent hash map can efficiently support a large number of readers and a bounded number of writers.

-APIs: p798

12.5.3 Atomic Update of Map Entries

-Note: Thread-safe data structure permits operations that are not thread-safe.

-Example: increment a count:



+Another thread might be updating the same count at the same time.

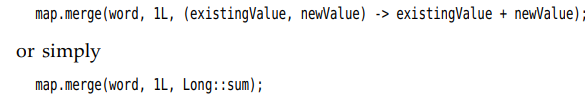
-Nowadays, Java API provide method that make atomic updates: compute():



+variants: computeIfPresent, computeIfAbsent that only compute a new value when there is already an old one, or when there isn’t yet one.



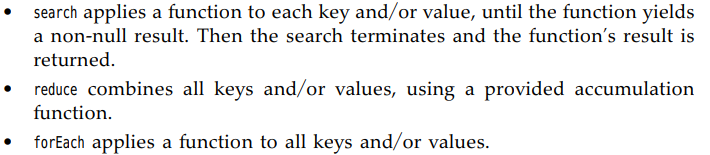
-Merge:



12.5.4 Bulk Operations on Concurrent Hash Maps

-Java API provides bulk operations on concurrent hash maps that can safely execute while other threads operate on the map. The bulk operation traverse the map and operate on the elements they find as they go along.

-There are 3 kinds of operations:



-There are 4 versions of operation:





-Each of the operations, you need to specify a **parallelism threshold**. If elements > threshold, the operation is parallelized. If the operation run in single thread, use threshold = Long.MAX\_VALUE, if the operation run in maximum threads, threshold=1

-Example: find the 1st words occurs > 1000

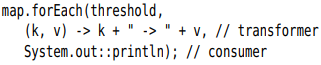


-forEach has 2 variants:

+1st applies a **consumer** function to each map entry:



+2nd applied the **transformer** then its result is passed to the consumer:



+The transformer can used as a filter, if the transformer return **null**, the value is **skipped**:



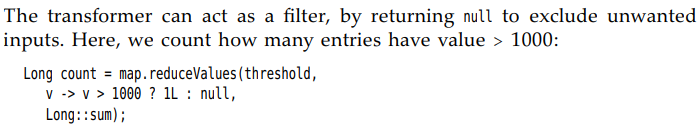
-reduce operation combine the inputs with an **accumulation** function.

+Compute the sum of all values:



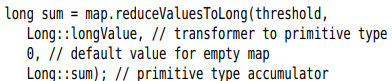
+You can supply a transformer. Example: compute the length of longest key:





+Note: If the map is empty, or all entries have been filtered out, reduce returns null. If there is only 1 element, transformer returned and accumulator is not applied.

-Primitive type: transform input to primitive value and specify a default value and an accumulator function. Default value is returned when the map is empty.



12.5.5 Concurrent Set Views

-You want a large, thread-safe set but there is no ConcurrentHashSet.

-Use **ConcurrentHashMap.newKeySet**:



12.5.6 Copy on Write Arrays

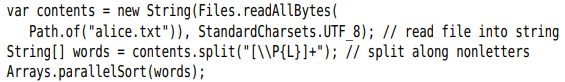
-CopyOnWriteArrayList and CopyOnWriteArraySet are thread-safe collections which all mutators make a copy of the underlying array.

-When thread iterate greatly outnumber the thread mutate the collection.

12.5.7 Parallel Array Algorithms

-Arrays class some parallelized operations

-**parallelSort()**



+You can supply a Comparator



+Supply the bound of a range:

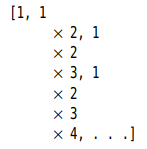


-**parallelSetAll()**:



-**parallelPrefix**():





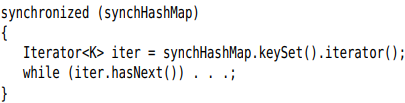
12.5.8 Older Thread-Safe Collections

-Any collection class can be made thread-safe by a **synchronization wrapper**



+The results are protected by a lock.

-**Client-side locking** (using the same lock that an object uses) need to use to iterate over collection while another thread has the opportunity to mutate it:



-You are usually better off using the collections in java.util.concurrent instead of the synchronization wrapper. One exception is an array list that is frequently mutated, use synchronized ArrayList.

-API: p800

# 12.6 Tasks and Thread Pools

-Construct a new thread is expensive because it involves inter-action with the OS.

-If the program creates a large number of short-lived threads, don’t map each task to a separate thread, use a **thread pool** instead.

- A **thread pool** contains a number of threads that are ready to run.

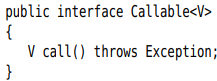
+You give a **Runnable** to the pool, one of the threads calls **run()**.

+When run() exits, the thread doesn’t die but stay around to serve the next request.

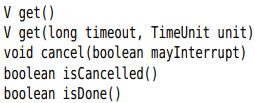
12.6.1 Callables and Futures

-A **Runnable** encapsulates a **task** that runs **asynchronously** with no parameters and no return value.

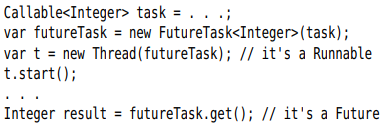
-A **Callable** is similar to Runnable, but it returns a value:



-A **Future** holds the **result** of an **asynchronous computation**:



-One way to **execute** a **Callable** is use a **FutureTask**: implements both Future and Runnable interface.



-Most commonly, you will pass a **Callable** to an **executor**.

-API:

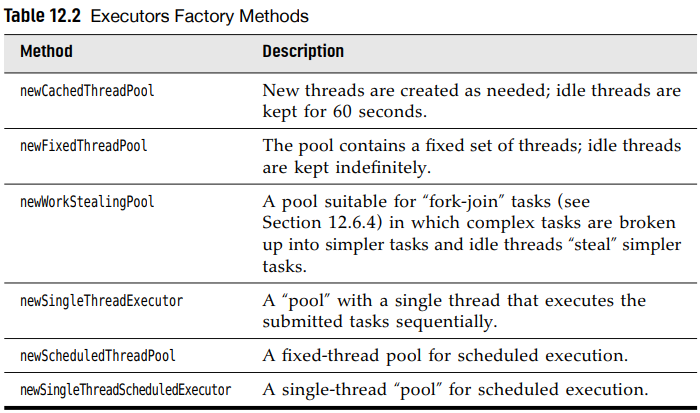
+Callable: run

+Future: get, cancel, isCancelled, isDone

+FutureTask: both Callable and Future

12.6.2 Executors

-**Executors** class has static factory methods to **construct** thread pools:



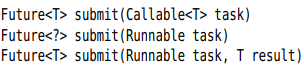
+Use a **cached thread pool** when you have threads that are short-lived or spend lots of time blocking.

+The number of **concurrent threads** = the number of **processor cores**. Use **fixed thread pool** to bound the number of concurrent threads.

+**single-thread executor** is useful for **performance** **analysis**.

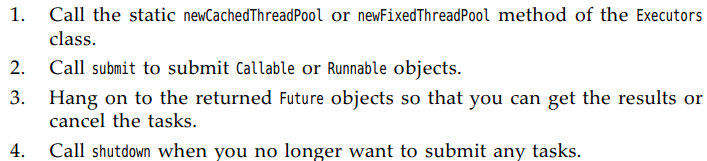
+These methods return an object of **ThreadPoolExecutor** class that implements **ExecutorService** interface.

-**Submit** a Runnable or Callable to an ExecutorService. The pool will run the submitted task



-When you are done with a thread pool, call **shutdown**() that initiate the shutdown sequence: when all tasks are finished, the threads in the pool die. Or use **shutdownNow**(): cancels all tasks that have not begun

-**Summary**: Thread pool do



-**ScheduledExecutorService** interface has methods for **scheduled** or **repeated** **execution** of **tasks**. It’s a generalization of java.util.Timer

+**newScheduledThreadPool** and **newSingleThreadScheduledExecutor** methods implement this interface.

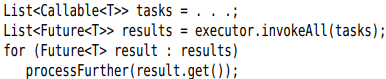
+You can **schedule** a Runnable or Callable to run once after an initial delay or run **periodically**.

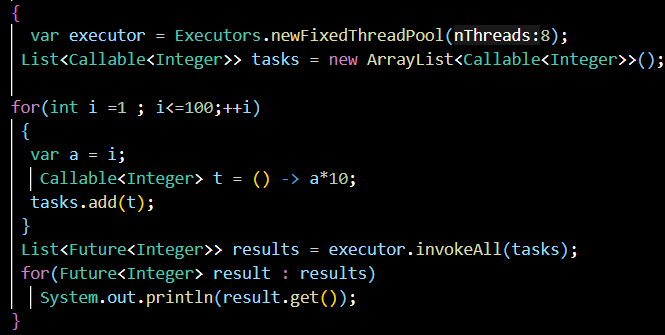
-API: p804

12.6.3 Controlling Groups or Tasks

-Sometimes, an executor is used for a more tactical reason – like control a **group** of related **tasks**. For example: shutdown()

-**invokeAll()**: submit all objects in a collection of Callable objects and return the result of a completed task.

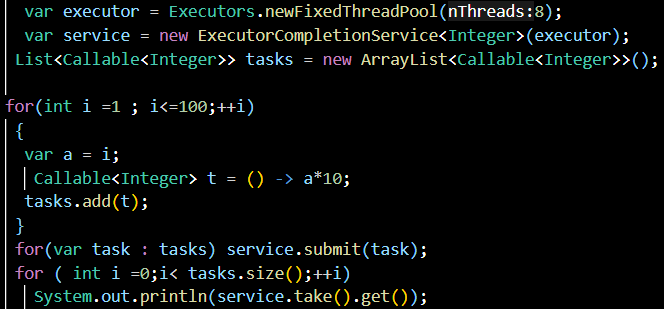




+The 1st reulst.get() blocks until the 1st result is available

-> get orderd.

-**ExecutorCompletionService**: submit each task to the completion service. It manages a blocking queue of Future objects, containing the results of submitted tasks as they become available -> efficient, but no order



-**invokeAny()**: parallelize the task. It terminates as soon as any task returns -> use for search the 1st verified thing

-**Tip**: In program, should use executor services to manage threads instead of launching threads individually.

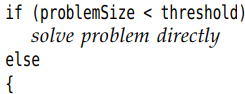
-API: p811

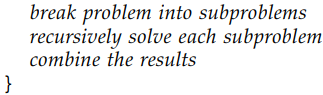
12.6.4 The Fork-Join Framework

-Some app use a large number of threads that are mostly idle. Example: web server uses one thread per connection.

-Other apps use on thread per processor core, to carry out computationally intensive tasks. (image/video processing…). The **fork-join framework** support this.

-Processing task can decompose into subtasks:

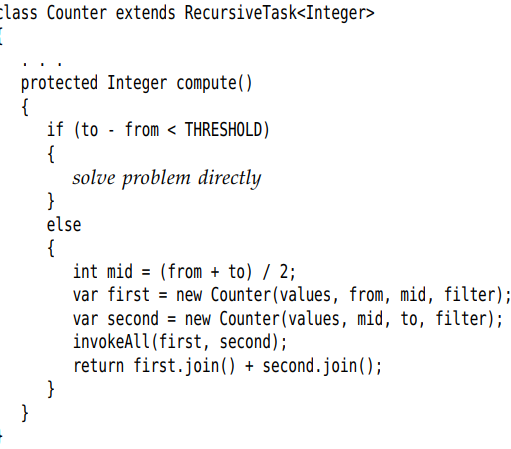




+Example: image processing: top half and bottom half. If you have enough idle processors, those operations can run in parallel.

-Suppose: count how many elements fulfill a property. We cut the array in half, compute each half, add them up

-Use **RecursiveTask<T>** or **RecursiveAction** to form a class that is usable by the framework:



+**invokeAll()** receives a number of tasks and blocks until all of them have completed. **Join()** yields the result.

# 12.7 Asynchronous Computations

-So far, our approach to concurrent computation has been to break up a task, then wait until all pieces have completed.

-Now we see how to implement wait-free or **asynchronous** computations.

12.7.1 Completable Futures

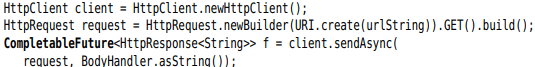
-Future object: get() to obtain the value, blocking until the value is available.

-**CompletableFuture** class implements the Future interface: you register a **callback** that will be invoked with the result once it is available.



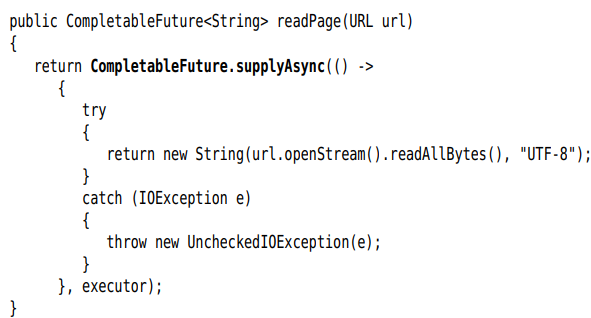
+So you can process the result without blocking one it is available.

-Some API methods return CompletableFuture objects. Example: fetch a web page asynchronously with **HttpClient** class (Vol 2 Chap 4):



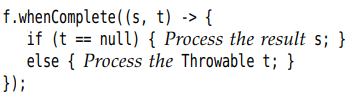
-Most of the time, you need to make CompletableFuture yourself. To run a task asynchronously and obtain a CompletableFuture, call static CompletableFuture.supplyAsync()

+Example: Read web page without HttpClient class



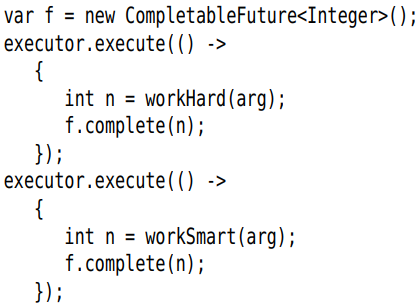
+The 1st argument is Supplier<T>, it is similar Callable<T> but cannot throw checked exception.

-A CompletableFuture can complete in 2 ways: a result or an uncaught exception. To handle both, use **whenComplete():**

****

-The CompletableFuture is **completable** because you can manually set a completion value. In other concurrency libraries, it called a **promise.**

-When create a CompletableFuture with supplyAsync, the completion is implicitly set when the task has finished. But the result can be set explicitly:



-Instead complete a future with an exception, call:



-**isDone()** tell that whether a Future object has been completed.

12.7.2 Composing Completable Furures

-Nonblocking calls are implemented through callbacks. The programmer registers a callback for the action that should occur after a task completes.

-CompletableFuture class provides a mechanism for **composing** asynchronous tasks into a processing pipeline

-Example: Extract all images from a web page:



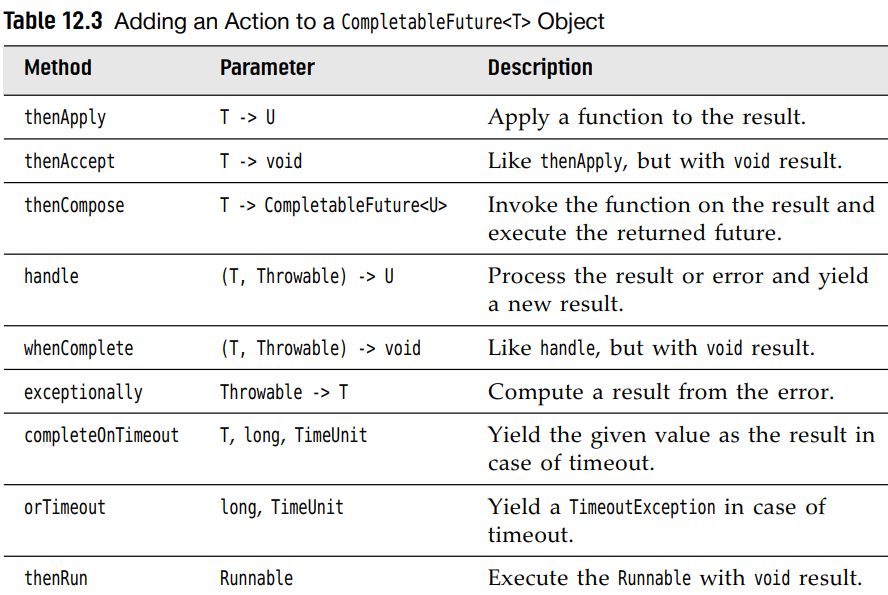




+The 1st future has completed, its results is fed to the getImageURLs()->final result.

-With completable futures, you just specify what to have done and in which order. All the code is in one place.

-Add an action to a CompletableFuture<T> object:



+**thenApply()**



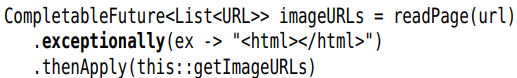
The 2nd call runs f in yet another thread.

+**thenCompose()**: Suppose read a web page from a URL





+**handle(), whenComplete(), exceptionally()** handle exceptions.



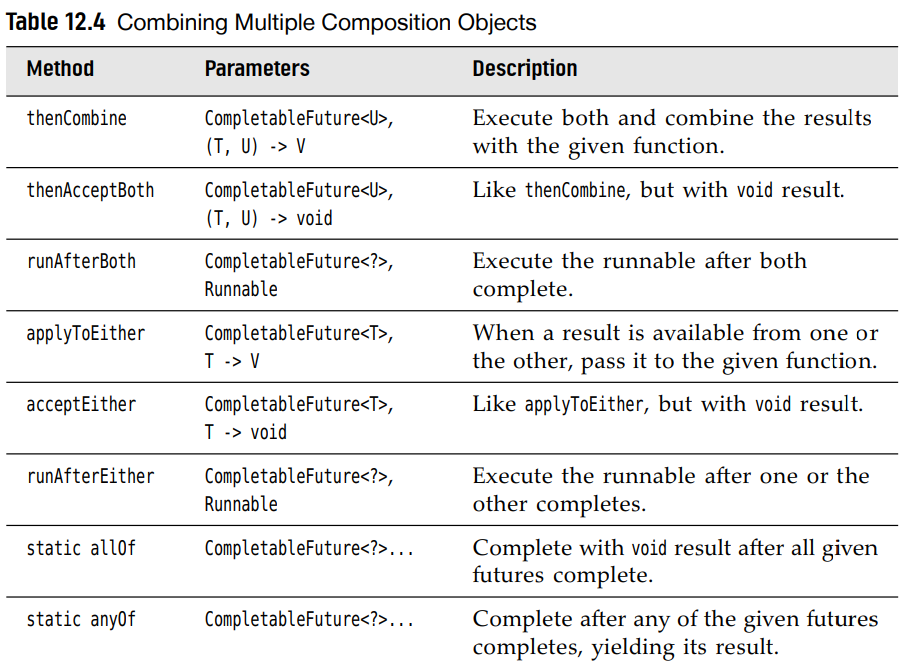
+**completeOnTimeOut()**:



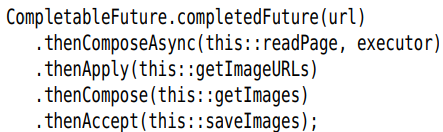
+**orTimeOut()** throw an exception on timeout:

****

**-**Combine multiple futures:



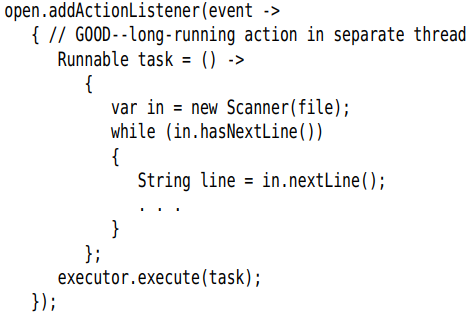
-To start the asynchronous computation, make a pipeline composing completed futures. Example:



12.7.3 Long-Running Tasks in User Interface Callbacks

-When the program needs to do sth time-consuming, you cannot do the work in the UI thread, so start another worker thread.

+Example: read a file when the user clicks a button



-But you can’t directly update the UI from the worker thread that executes the long-running task. You need to schedule any UI updates to happen on the UI thread. Each user interface library provides some mechanism to schedule a Runnable for execution on UI thread.



-You specify actions for long-running task (runs on a separate thread), and progress updates and the final disposition (runs on UI thread)

-The typical UI activities of a background task

+After each work unit, update the UI to show progress

+After the work is finished, make a final change to UI

(Learn later)

# 12.8 Processes

-Up to now, we’ve seen how to execute Java in separate threads within the same program. Sometimes you need to execute another program. Use **ProcessBuilder** and **Process** classes:

+**Process** class executes a command in a separate OS process and lets you interact with its standard input, output and error streams.

+**ProcessBuilder** class configures a Process object.

12.8.1 Building a process

-Specify the **command** that you want to execute. Supply a List<String> or simply the strings

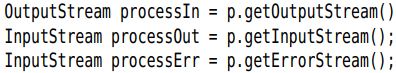


+Caution: The 1st string must be an executable command. For example, to run dir command in Windows: build a process with string “cmd.exe”,”/C”,”dir”

-Each process has a **working directory**, which is used to resolve relative directory names. By default, a process have the same working directory as the VM. You can change it with: 

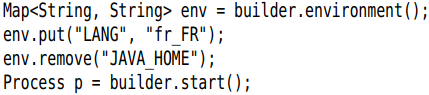
+Note: Shortly start a process 

-Specify what should happen to the standard input, output and error streams of the process. By default, each is a pipe that you can access:



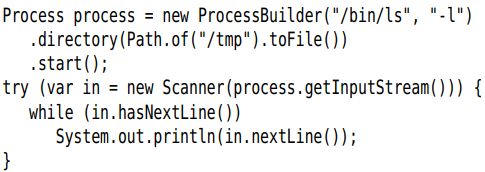
+Note that the input stream of the process is an output stream in JVM.

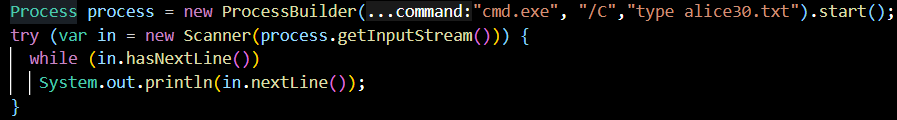
-Modify the environment variables of the process.



12.8.2 Running a Process

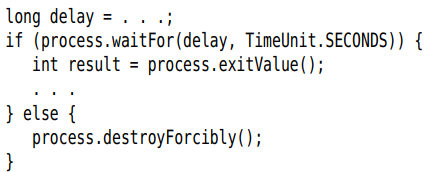
-start()





-Wait for the process to finish: 

+If you don’t want to wait indefinitely



-Instead of waiting for the process to finish, you can use isAlive() to see whether it is still alive.

-To kill the process: destroy() destroyForcibly()

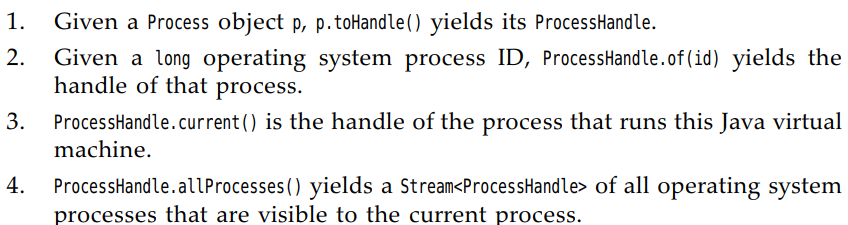
-Receive asynchronous notification when the process has completed:



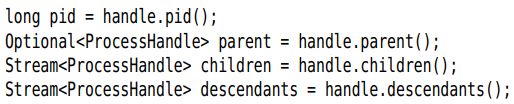
12.8.3 Process Handles

-To get more information about a process that your program started, or any process that is currently on the your machine, use **ProcessHandle** interface.

-You can obtain a ProcessHandle in 4 ways:

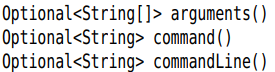


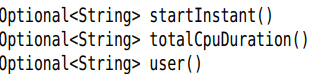
-Given a process handle, you can get its process ID, its parent process, its children, and descendants:



+Note: The Stream<ProcessHandle> returned by allProcesses(), children(), descendants() are just snapshots in time.

-info() yields a ProcessHandle.info object with methods to obtain information about the process:





-Terminate: isAlive(), supportNormalTermination(), destroy(), destroyForcibly(), onExit() as the Process class

-API: p836