

Topic 9

Intro to Concurrency and Multithreading

Learning Outcomes

- **Describe** when and how *processes* are created and destroyed and what information they encapsulate
- **Describe** when and how *threads* are created and destroyed and what information they encapsulate
- **Describe** how threads are scheduled and what control the programmer has over this process
- **Program** threads by extending the `java.lang.Thread` class
- **Program** threads by implementing the `java.lang.Runnable` interface
- **Debug** threads using the eclipse IDE
- **Describe and Document Diagrammatically** Java thread states and their possible transitions

Introduction: Processes

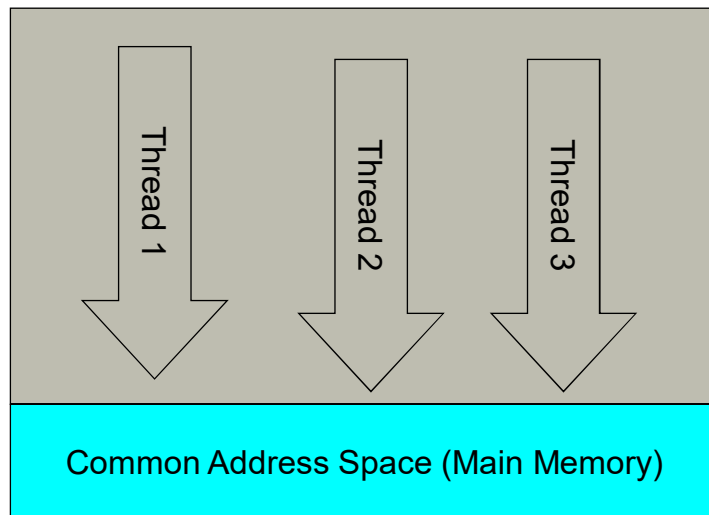
- Modern operating systems manage the concurrent scheduling of multiple processes
- Modern CPUs have multiple cores (effectively representing multiple CPUs)
 - can concurrently execute multiple processes and threads
- Generally each user program operates within its own **process** encapsulating the following information:
 - the process state (e.g. ready, running, waiting, stopped)
 - the program counter (PC) containing the address of the next executable instruction
 - saved CPU register values
 - memory management information (e.g. page tables, swap files)
 - I/O information e.g. file or socket descriptors, pending I/O requests etc.
- Processes are expensive to create and destroy so lightweight **threads** of execution within a process are used for concurrent operations within a single executable process

Introduction: Threads

- A **thread** is a unit of execution control within a process
 - i.e. a method execution within an OO language
- When the program starts it executes the main thread
 - e.g. the `main()` method in Java
- In a multithreaded program the main thread creates other threads which execute other methods concurrently
 - use the interface `java.lang.Runnable` and class `java.lang.Thread`
- Each thread:
 - has a method call stack (which includes method parameters and local variables)
 - a copy of CPU registers including the stack pointer (SP) and program counter (PC) (this is invisible to the Java program itself)
 - **shares** all of the process context described previously (i.e. code, data, memory management, I/O handles etc.)
- The operating system is responsible for scheduling processes and threads (context switching) using various policies (e.g. LRU, LFU etc.)
 - Context switching (scheduling) of threads is more efficient than processes though processes have the advantage of being able to execute on multiple machines and are more resistant to crashes
 - User programs should not be dependent upon scheduling policy to achieve correct results
- See `sadi.topic9.print.SingleThreaded`

Illustration

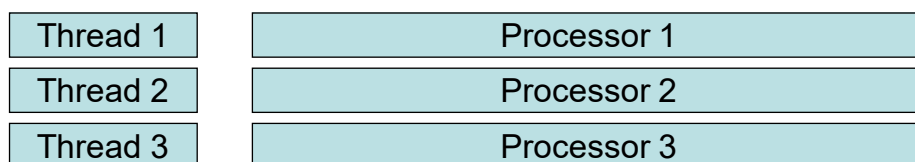
Threads (light weight processes within a process), sharing common address space.



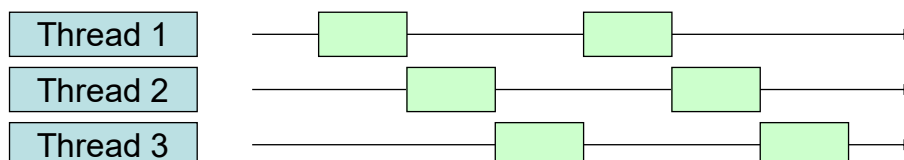
How do threads execute ?

In systems with many processors or cores threads MAY BE executed simultaneously on separate CPUs or CPU cores.

Intel has a process called hyper threading where some thread concurrency can exist on a single core.



In single processor/single core system threads ALWAYS share CPU time:

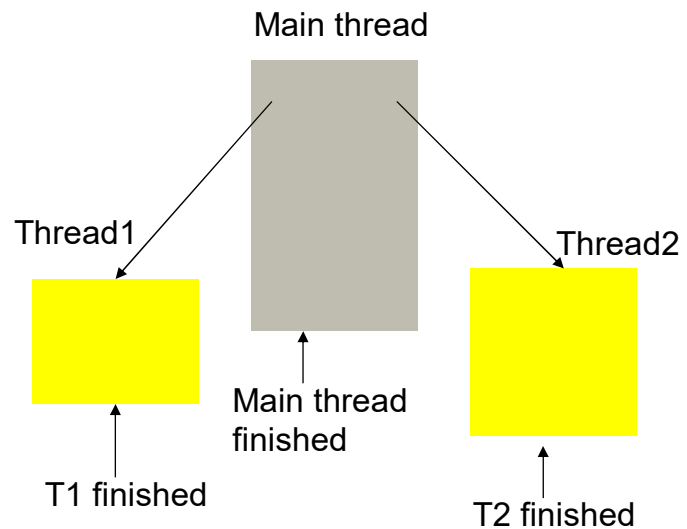


Single Threaded vs Multi-Threaded Programs

Single Threaded

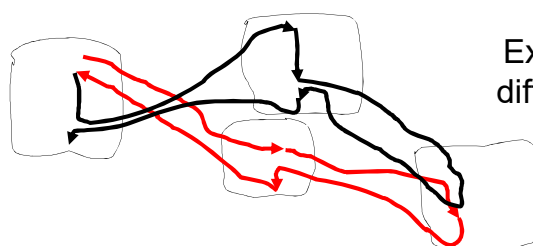


Multiple Threaded



How do threads execute?

- More than one stream of program execution running in parallel or concurrently.
- CPU executes a list of statements in each thread until all threads are complete.
- Along the way threads may call various methods and potentially access data (encapsulation?) from different objects.
- Two threads may attempt to access the same method or data at the same time causing an unpredictable outcome.
- This is addressed with thread synchronization (left to future courses)
- E.g. `Object.wait()/notify()` or `java.util.concurrent.*`;

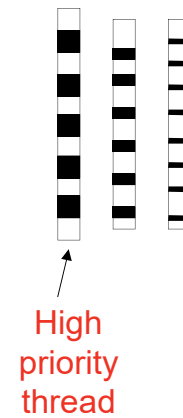


Execution paths through different objects/methods

Why use Threads?

- It makes sense as most of the time CPU is idle waiting for slower devices (e.g. I/O such as file, network or device) or input from user (human or other system).
- Some tasks (such as garbage collection of unused memory) can be done by a thread that has a lower priority. The higher the priority the greater the amount of CPU time allocated (depends on OS).
- Many applications (such as web or file servers) need to serve more than one client at a time. Java allows a server program to start a new thread for each new client.
- Some tasks can be done in parallel i.e. let the user work on something else rather than wait if possible
- Some frameworks (e.g. AWT/Swing) require certain tasks to be performed on specific threads

Multiple threads sharing CPU time doing multiple tasks



Creating threads by extending the Thread class

- The **Thread** class provides constructors and methods for creating and controlling the threads.
- The **run()** method of the **Thread** class specifies the code to execute when the thread is started.
- Hence the user creating a thread can derive a subclass of **Thread** and override the run method.
- This class has a method named **start()** which creates a separate thread and calls its **run()** method
- You can only run a thread once i.e. cannot restart it (see `Thread.start()` in API docs)
- We will now look at a sample program that uses two threads (of the same type)
- See `sadi.topic9.print.PrintThreads`

Simple Program using Thread class

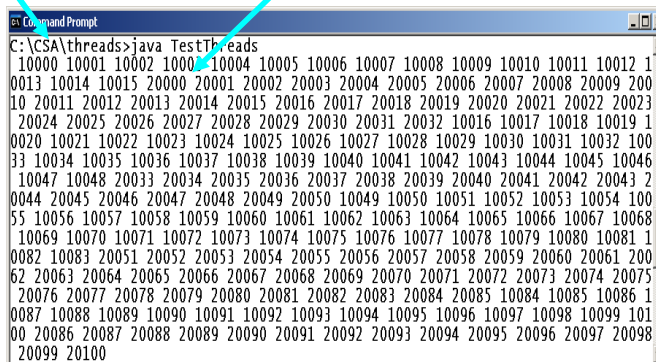
- The program has two independent threads. The first thread prints the numbers 10000 to 10100 in steps of 1. The second thread 20000 to 20100.
- Run this in the **debugger** and inspect the threads (you can right click on a breakpoint in Eclipse and use Breakpoint Properties->Filtering to apply to a specific thread)
- You can set processor affinity from task manager to force on to a single CPU core
- Note that the sequence of number printing is not deterministic .. Why?

1st thread printing

sequence starting from

10000

2nd thread printing sequence starting from 20000



```
C:\CSA\threads>java TestThreads
10000 10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 1
0013 10014 10015 20000 20001 20002 20003 20004 20005 20006 20007 20008 20009 200
10 20011 20012 20013 20014 20015 20016 20017 20018 20019 20020 20021 20022 20023
20024 20025 20026 20027 20028 20029 20030 20031 20032 10016 10017 10018 10019 1
0020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030 10031 10032 100
33 10034 10035 10036 10037 10038 10039 10040 10041 10042 10043 10044 10045 10046
10047 10048 20033 20034 20035 20036 20037 20038 20039 20040 20041 20042 20043 2
0044 20045 20046 20047 20048 20049 20050 10049 10050 10051 10052 10053 10054 100
55 10056 10057 10058 10059 10060 10061 10062 10063 10064 10065 10066 10067 10068
10069 10070 10071 10072 10073 10074 10075 10076 10077 10078 10079 10080 10081 1
0082 10083 20051 20052 20053 20054 20055 20056 20057 20058 20059 20060 20061 200
62 20063 20064 20065 20066 20067 20068 20069 20070 20071 20072 20073 20074 20075
20076 20077 20078 20079 20080 20081 20082 20083 20084 20085 10084 10085 10086 1
0087 10088 10089 10090 10091 10092 10093 10094 10095 10096 10097 10098 10099 101
00 20086 20087 20088 20089 20090 20091 20092 20093 20094 20095 20096 20097 20098
20099 20100
```

PrintThreads.java Program Features

- The **PrintThread** is an indirect subclass of **Thread**.
 - Its constructor takes one argument, the initial value for sequence, which is stored in instance variable **inital**.
 - This class has overridden the **run()** method to print the first 101 numbers starting from **inital**.
 - The main method has created two instances of the **PrintThread** classes passing different initial values.
 - Calling the **start()** method of these instances cause a new thread to be created in JVM which calls the **run()** method resulting in the two sequences being printed.
-
- Q. What must be done to print first 101 numbers of another sequence starting with 50000 ? _____
 - Q. What if **run()** is called directly? _____

Creating threads using the Runnable interface

- If you want to make an existing class run in a thread, you can use the **Runnable** interface, which contains a single method **run()**
- To make the existing class **A** run as a thread create a subclass of **A** that implements **Runnable** as in:

```
class RunnableA extends A implements Runnable
{
    public RunnableA(...) {...
    public void run() { ...
    ...
}
```
- Could be an inner class (even anonymous) but watch cohesion!
- Then create an instance of this subclass passing its reference to **Thread** class as in:


Thread tA = new Thread(new RunnableA(...))

Sample Program using Runnable interface

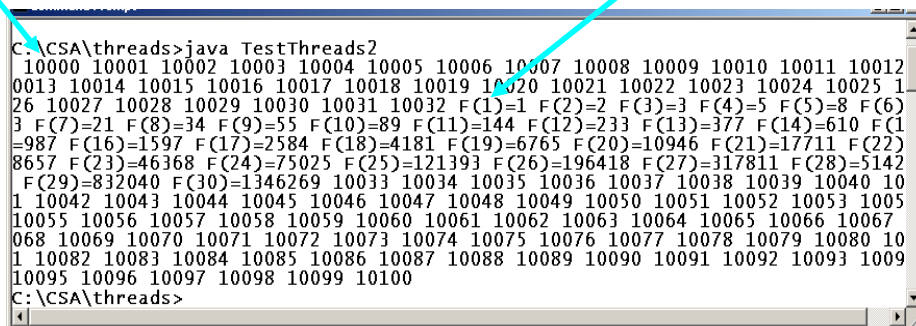
- Suppose that we have an existing class **Fibo** that prints the Fibonacci numbers.
- We want to modify our previous program to allow one thread to print the sequence as before (starting at 10,000) and another to print the Fibonacci numbers
- Rather than creating a class from scratch, we want to extend the existing class **Fibo** to run in a thread. We do this by creating a subclass of **Fibo** that implements the **Runnable** interface.

Sample Program using Thread class and Runnable interface

- The program below has two independent threads belonging to different classes. The first thread prints the numbers 10000 to 10100 in steps of 1 using the class **PrintThread** as before. The second thread prints the first 30 terms of the Fibonacci series using a different class **RunnableFibo**.
- The cpu has taken turns to print the two sequences as before.
- See `sadi.topic9.print.RunnableThreads`

1st thread printing
sequence starting from
10000

2nd thread printing the first 30
terms of the Fibonacci series



```
C:\CSA\threads>java TestThreads2
10000 10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012
0013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 1
26 10027 10028 10029 10030 10031 10032 F(1)=1 F(2)=2 F(3)=3 F(4)=5 F(5)=8 F(6)
3 F(7)=21 F(8)=34 F(9)=55 F(10)=89 F(11)=144 F(12)=233 F(13)=377 F(14)=610 F(1
=987 F(16)=1597 F(17)=2584 F(18)=4181 F(19)=6765 F(20)=10946 F(21)=17711 F(22)
8657 F(23)=46368 F(24)=75025 F(25)=121393 F(26)=196418 F(27)=317811 F(28)=5142
F(29)=832040 F(30)=1346269 10033 10034 10035 10036 10037 10038 10039 10040 10
1 10042 10043 10044 10045 10046 10047 10048 10049 10050 10051 10052 10053 1005
10055 10056 10057 10058 10059 10060 10061 10062 10063 10064 10065 10066 10067
10068 10069 10070 10071 10072 10073 10074 10075 10076 10077 10078 10079 10080 10
1 10082 10083 10084 10085 10086 10087 10088 10089 10090 10091 10092 10093 1009
10095 10096 10097 10098 10099 10100
C:\CSA\threads>
```

RunnableThreads.java Program Features

- PrintThread** is a subclass of **Thread** as before
- Fibo** is an existing class with one method named `getNext()` to return the next term in the series
- The class **RunnableFibo** extends **Fibo** and implements **Runnable**. Note the overridden `run()` method uses the `getNext()` of the superclass
- The `main()` method creates **PrintThread** object as before to print the sequence numbers 10,000 to 10,100
- The `main()` method also creates an instance of **RunnableFibo** and passes it to a **Thread** constructor, allowing Fibonacci numbers to be printed in a thread
- Calling `start()` method creates the threads and invokes `run()` indirectly

Quiz: How do I pass arguments to the run method?

- Suppose I want only Fibonacci numbers greater than `n` to be printed, where `n` is a variable.
- How can I pass `n`?

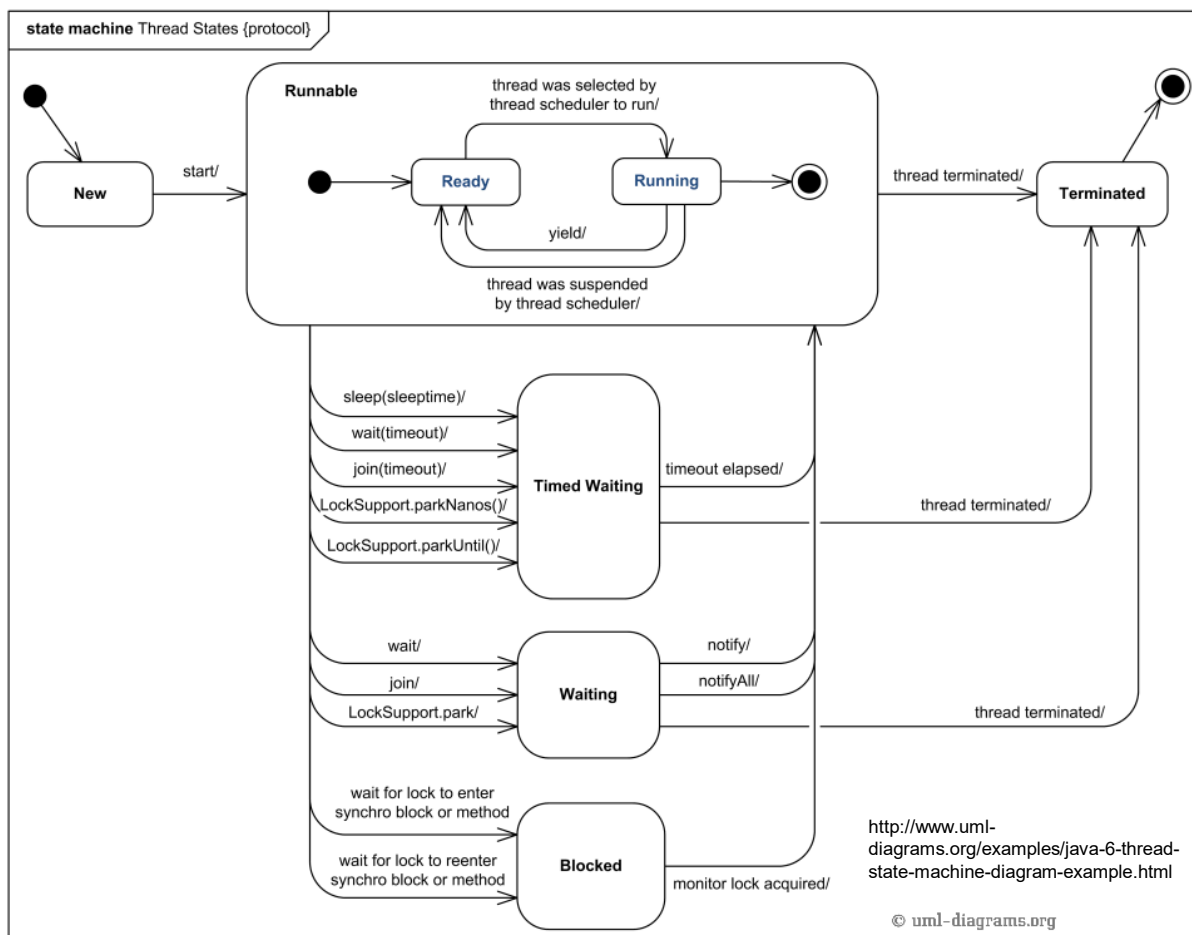
Now let us have our first look at the API documentation for `java.lang.Thread`

`setDaemon` is an interesting method since the JVM will terminate when all non-daemon threads have finished

i.e. don't set a thread to daemon if you want it to keep the application alive!

Thread States

- Because Threads are subject to scheduling based on resource availability they have a lifecycle involving a number of states
- The Java Thread lifecycle is represented diagrammatically on the following slide
- The states are represented by the enum `java.lang.Thread.State` and returned by `Thread.getState()`
- When a thread is created (instantiated) but not yet started (via `Thread.start()`) it is placed in the NEW state
- When the thread is started (via `Thread.start()`) it is placed in the RUNNABLE state
- While a thread is RUNNABLE its sub states are either RUNNING when CPU time is allocated to it or READY when CPU time expires and another thread is scheduled in its place or the thread calls `Thread.yield()`
 - NOTE: the sub states are OS/architecture dependent and are not directly visible to the Java program
- The thread enters the WAITING, TIMED_WAITING or BLOCKED states as a result of explicit calls such as `Thread.sleep()`, `Thread.join()` or `Object.wait()` or if it is waiting for an I/O operation to finish. Details on the following slide.
- When a thread has finished (the `run()` method terminates) it enters the TERMINATED state



Examples: What will be the output ? Why ?

- Modify one thread using `Thread.yield()`
 - see `sadi.topic9.print.PrintThreadsYield`
 - set a breakpoint before starting threads use processor affinity in Windows Task Manager (Details Tab) to limit to single CPU to accentuate effect
 - Windows **Resource Monitor** and **Performance Monitor** are useful for inspection
 - will usually use priorities instead
- Modify the priority of one thread using `Thread.setPriority()`
 - see `sadi.topic9.print.PrintThreadsPriority`
 - breakpoint and affinity as above to see effect on multicore
- First thread sleeps after printing 10050
 - see `sadi.topic9.print.PrintThreadsSleep`
- Using the method `Thread.isAlive()`
 - see `sadi.topic9.print.PrintThreadsIsAlive`
- Using the method `join` at 20050
 - see `sadi.topic9.print.PrintThreadsJoin`

NOTE: Scheduling related methods such as `yield()`, `sleep()` etc. are OS, JVM and processor/architecture dependent

Swing and Threading

- Time-consuming tasks should be run in a separate *Worker Thread* (not on the *Event Dispatch Thread* i.e. in a listener)
 - otherwise the application hangs or becomes unresponsive (you may have seen this in assignment 2!)
- Swing components should ONLY be accessed on the *Event Dispatch Thread*
 - they can be created by the main thread (from `main()`) provided this is the last thing that is done otherwise use `javax.swing.SwingUtilities.invokeLater()` or `invokeAndWait()` methods
 - there are some exceptions i.e. some Swing components are Thread safe (see API for details)
- `javax.swing.SwingWorker` simplifies this process
 - see following slide and example in package `sadi.topic9.swing.thread`

Multi-Threaded Code Example for AWT Swing

To call a *worker* method on a separate thread ..

```
new Thread()  
{  
    @Override  
    public void run()  
    {  
        //call worker methods on separate thread  
    }  
}.start();
```

to update the GUI from a non UI thread ..

```
SwingUtilities.invokeLater(new Runnable()  
{  
    @Override  
    public void run()  
    {  
        // do GUI update on UI thread  
    }  
});
```

The javax.swing.SwingWorker helper class

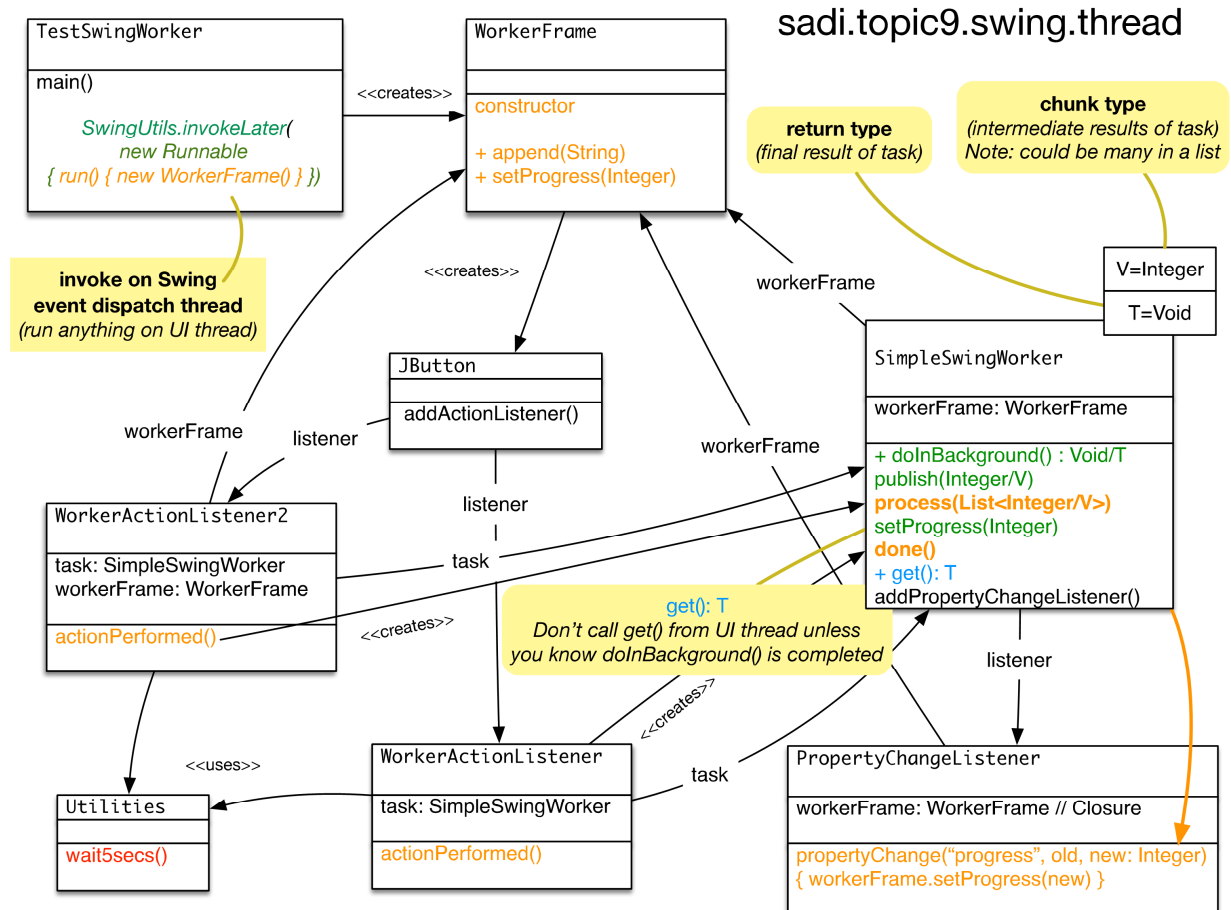
This abstract class provides support for handling time consuming tasks with progress updates by managing the various threads as follows:

- **Current thread**
 - the thread on which `SwingWorker.execute()` is called
 - often an event handler thread since the event initiates the work to be done
 - returns immediately or can wait for the `SwingWorker` to complete using the get methods
 - like `Threads` can only be executed once per instance
- **Worker thread**
 - the `SwingWorker.doInBackground()` method is automatically called on this thread following `SwingWorker.execute()`
 - by default there are two bound properties available (state and progress)
 - use shortcut method `setProgress()` to fire a property change on the “progress” property
 - or use `firePropertyChange()` or `getPropertyChangeSupport()` to notify these or other custom `PropertyChangeListener`s
 - call `publish()` to process updates as partial work is done (e.g. a line is read from a file) etc.

The javax.swing.SwingWorker helper class

- **Event Dispatch Thread**
 - all Swing related activities occur on this thread
 - `SwingWorker` invokes its `process()` and `done()` methods and notifies any `PropertyChangeListener` on this thread
 - NOTE: `done()` method (an override) is called automatically when `doInBackground()` is finished
 - `process(List<V> chunks)` receives calls from the `publish()` method (called from the worker thread) asynchronously on the *Event Dispatch Thread*
 - calls to `publish()` may be coalesced for efficiency (combined) which is why the parameter is a variable length arg list
 - usually the *Current* thread is the *Event Dispatch Thread*

See `sadi.topic9.swing.thread.TestSwingWorker` *example*



Starting and Stopping Threads

- A process (application exits when all non-daemon threads have finished) i.e. if the only threads remaining are daemon threads the application will exit
 - see `java.lang.Thread.setDaemon(boolean on)`
- You have already seen how to start Threads using the `Thread.start()` method
- You can stop methods externally using `Thread.stop()` however this is dangerous and *deprecated*
- Therefore threads should either terminate naturally (when `Thread.run()` exits) or handle external calls to `Thread.interrupt()`
- Calling `interrupt` sets the state of `Thread.isInterrupted()` to true (which can be checked from the worker thread) **OR** raises various exceptions such as `InterruptedException` (see `Thread.interrupt()` in API for details)
- See examples in package `sadi.topic9.swing.thread.startstop`

References

- <http://docs.oracle.com/javase/tutorial/essential/concurrency/>
- <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/>