# Confinement - Semaphores

# (Instance) Confinement

- Want to use thread unsafe objects in a thread safe way.
- Encapsulate in wrapper object.
  - Encapsulation makes it easier to ensure proper lock held when manipulating the wrapped object.
  - Proxy pattern example:
    - Same interface - just controlled for concurrency
    - Guarded methods in proxy interface
    - Explicit guards on unsafe object's lock
  - Or wrapper could be abstraction built on unsafe class
    - Thread safe course with enrollment based on unsafe HashSet.
    - Thread safe queue based on unsafe ArrayList.

**What design pattern is used & how might it support safe access?**

- Java "monitor" pattern.
- Note - can be the default lock or any other one used consistently.

# Confinement Example - Implicit Lock

```
public class Dictionary {
  private final HashMap<String, String> dict =
    new HashMap<String, String>() ;

  public synchronized String getDef(String word) {
    return dict.get(word) ;
  }

  public synchronized void addDef(String word, String def) {
    if ( dict.get(word) == null ) {
      dict.put( word, def ) ;
    }
  }

  public synchronized Map<String, String> getMap() {
    return Collections.unmodifiableMap( dict ) ;
  }
}
```

**What is being used as the lock in this example?**

# Confinement Example - Explicit Lock

```
public class Dictionary {
  private final HashMap<String, String> dict =
    new HashMap<String, String>() ;

  private Object lock = new Object() ;

  public String getDef(String word) {
    synchronized(lock) {
      return dict.get(word) ;
    }
  }

  public void addDef(String word, String def) {
    synchronized(lock) {
      if ( dict.get(word) == null ) {
        dict.put( word, def ) ;
      }
    }
  }

  public Map<String, String> getMap() {
    synchronized(lock) {
      return Collections.unmodifiableMap( dict ) ;
    }
  }
}
```

**Is there an advantage to using a explicit vs an implicit lock?**

# ~~Synchronized~~

- We won't be using synchronized this semester
  - This means you will receive 0 credit if you use it in your assignments/projects
- Why?
  - You already know how to use synchronized
  - It hides a lot of details which can make working with it more complicated
  - There are many better, more specialized guards in the concurrency libraries

# Semaphore

- Found in `java.util.concurrent.Semaphore`
- A counting semaphore that maintains permits
- `acquire()` – get at permit
- `release()` – give up a permit
- When all permits are taken, any thread attempting to acquire a new one will block until another thread releases a permit
- This can be used to easily control how many threads have access to a shared resource
- **Signaling Semaphore** – Initial permits set to 0
- **Mutex Semaphore** – Initial permits set to 1

# Confinement Example - Explicit Lock

```
public class Dictionary {
    private final HashMap<String, String> dict =
        new HashMap<String, String>() ;

    private Semaphore mutex = new Semaphore (1);

    public String getDef(String word) throws InterruptedException {
        mutex.acquire ();
        String value = dict.get(word) ;
        mutex.release ();
        return value;
    }

    public void addDef (String word, String def) throws InterruptedException {
        mutex.acquire ();
        if ( dict.get(word) == null ) {
            dict.put( word, def ) ;
        }
        mutex.release ();
    }

    public synchronized Map<String, String> getMap() throws InterruptedException {
        mutex.acquire ();
        Map map =  Collections.unmodifiableMap( dict ) ;
        mutex.release ();
        return map;
    }
}
```

# Additional Methods

- Semaphore has several additional methods besides acquire and release
- You should take a few minutes to investigate before attempting to use it