

# Lecture 10:

## Java Threads Programming 3

NCCU 高等軟體設計

Fall 2005

Nov. 29, 2005

# Concurrency Patterns

- Producer-Consumer
- Readers-Writers
- Worker Threads
- Future Objects
- Active Objects
- Two-Phase Termination
- ThreadLocal

# Future Object Pattern (提貨卷)

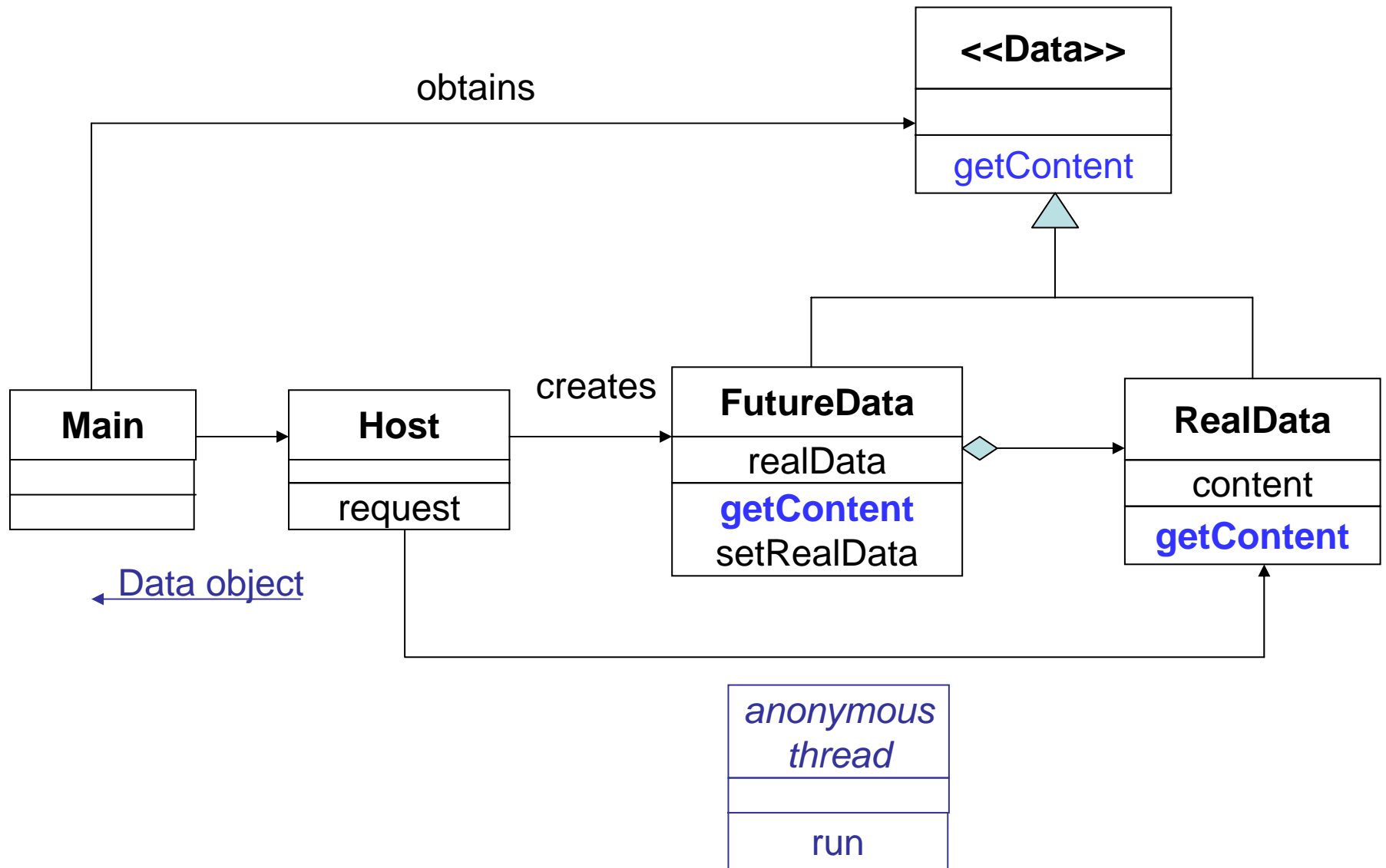
呼叫 <> 立即執行

- Synchronous communications
  - Client calls the server's method and
  - *Server creates a thread to handle the request*
  - *Client waits for the server to return the results*
- Asynchronous communications
  - Client calls the server's method and
  - *Server creates a thread to handle the request*
  - *Client gets a "future" object and returns*
  - Client may do other work first and then
  - *Client requests the results from the future object*

# Example of Future Pattern

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main BEGIN");  
        Host host = new Host();  
        Data data1 = host.request(10, 'A'); // future object // 連印10個A  
        Data data2 = host.request(20, 'B'); // future object  
        Data data3 = host.request(30, 'C'); // future object  
  
        System.out.println("main otherJob BEGIN");  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException e) {  
        }  
        System.out.println("main otherJob END");  
  
        System.out.println("data1 = " + data1.getContent()); // request the result  
        System.out.println("data2 = " + data2.getContent()); // request the result  
        System.out.println("data3 = " + data3.getContent()); // request the result  
        System.out.println("main END");  
    }  
}
```

# Participants & Class Diagram



```

public class Host {
    public Data request( final int count, final char c ) {
        System.out.println("  request(" + count + ", " + c + ") BEGIN");

        // (1) 建立FutureData的實體
        final FutureData future = new FutureData();

        // (2) 爲了建立RealData的實體，啓動新的執行緒
        new Thread() {
            public void run() {
                RealData realdata = new RealData(count, c);           //Anonymous class
                future.setRealData(realdata);
            }
        }.start();

        System.out.println("  request(" + count + ", " + c + ") END");

        // (3) 取回FutureData實體，作爲傳回值
        return future;
    }
}

```

```
public interface Data {  
    public abstract String getContent();  
}
```

```
public class FutureData implements Data {  
    private RealData realdata = null;  
    private boolean ready = false;  
    public synchronized void  
        setRealData(RealData realdata) {  
        if (ready) {  
            return;    // balk  
        }  
        this.realdata = realdata;  
        this.ready = true;  
        notifyAll();  
    }  
    public synchronized String getContent() {  
        while (!ready) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        return realdata.getContent();  
    }  
}
```

```
public class RealData implements Data {  
    private final String content;  
    public RealData(int count, char c) {  
        System.out.println(  
            " making RealData(" + count + ", " + c + "  
            BEGIN");  
        char[] buffer = new char[count];  
        for (int i = 0; i < count; i++) {  
            buffer[i] = c;  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
            }  
        }  
        System.out.println(  
            " making RealData(" + count + ", " + c + "  
            END");  
        this.content = new String(buffer);  
    }  
    public String getContent() {  
        return content;  
    }  
}
```

# Discussion

- Does the “*request*” method of Host class need to be synchronized?
  - No, why? Local variables are thread-safe.

```
public class Host {
    public Data request( final int count, final char c ) {
        System.out.println("  request(" + count + ", " + c + ") BEGIN");
        final FutureData future = new FutureData();
        new Thread() {
            public void run() {
                RealData realdata = new RealData(count, c);
                future.setRealData(realdata);
            }
        }.start();
        System.out.println("  request(" + count + ", " + c + ") END");
        return future;
    }
}
```

# Exception Handling in Future Object

- What if there is an exception raised in the anonymous *thread*?

```
new Thread() {  
    public void run() {  
        RealData realdata = new RealData(count, c);  
        future.setRealData(realdata);  
    }  
}.start();
```

Exception  
raised



- Where to catch the exception? Nowhere!  
The program gets stuck!

```
public class Host {  
    public Data request(final int count, final char c) {  
        System.out.println("    request(" + count + ", " + c + ") BEGIN");
```

```
        // (1) 建立FutureData的實體
```

```
        final FutureData future = new FutureData();
```

```
        // (2) 爲了建立RealData的實體，啓動新的執行緒
```

```
        new Thread() {  
            public void run() {  
                try {  
                    RealData realdata = new RealData(count, c);  
                    future.setRealData(realdata);  
                } catch (Exception e) {  
                    future.setException(e);  
                }  
            }  
        }
```

```
        }.start();
```

```
        System.out.println("    request(" + count + ", " + c + ") END");
```

```
        // (3) 取回FutureData實體，作爲傳回值
```

```
        return future;
```

```
    }
```

```
}
```

```

import java.lang.reflect.InvocationTargetException;
public class FutureData implements Data {
    private RealData realdata = null;
    private InvocationTargetException exception = null;
    private boolean ready = false;
    public synchronized void setRealData(RealData realdata) {
        if (ready) { return; }
        this.realdata = realdata;
        this.ready = true;
        notifyAll();
    }
    public synchronized void setException(Throwable throwable) {
        if (ready) { return; }
        this.exception = new InvocationTargetException(throwable);
        this.ready = true;
        notifyAll();
    }
    public synchronized String getContent() throws InvocationTargetException {
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        if (exception != null) { throw exception; }
        return realdata.getContent();
    } }

```

# 補充：Java Inner Classes

# Inner Classes

- Inner, or Nested, classes are standard classes declared within the scope of a standard top-level class.
- There are different kinds of inner class
  - nested top-level class
  - member class
  - local class
  - anonymous class

# Nested Top-Level Classes

```
class outer {  
    private static class NestedTopLevel {  
        normal class stuff  
    }  
    normal class stuff  
}
```

- Nested top-level classes are **declared static** within a top-level class (sort of like a *class* member).
- They follow the same rules as standard classes
  - `private static` classes cannot be seen outside the enclosing class
  - `public static` allows the class to be seen outside

# LinkedStack2

```
public class LinkedStack2 {
    private StackNode tos = null;

    private static class StackNode {
        private Object data; private StackNode next, prev;

        public StackNode( Object o ) { this( o, null ); }
        public StackNode( Object o, StackNode n ) {
            data = o; next = n;
        }

        public StackNode getNext() { return next; }
        public Object getData() { return data; }
    }

    public boolean isEmpty() { return tos == null; }
    public boolean isFull() { return false; }
    public void push( Object o ) { tos = new StackNode( o, tos ); }
    public void pop() { tos = tos.getNext(); }
    public Object top() { return tos.getData(); }
}
```

# Member Classes

- A member class is a *nested top-level* class that is **not declared static**.
- This means the member class has a `this` reference which refers to the enclosing class object.
- Member classes cannot declare static variables, methods or nested top-level classes.
- Member objects are used to create data structures that need to know about the object they are contained in.

# Class5

```
class Class5 {  
  
    private class Member {  
        public void test() {  
            i = i + 10;  
            System.out.println( i );  
            System.out.println( s );  
        }  
    }  
  
    public void test() {  
        Member n = new Member();  
        n.test();  
    }  
  
    private int i = 10;  
    private String s = "Hello";  
  
    public static void main(String args[])  
    {  
        Class5 five = new Class5();  
        five.test();  
    }  
}
```

>Java Class5

20

Hello

>

# this Revisited

- To support member classes several extra kinds of expressions are provided
  - `x = this.dataMember` is valid only if `dataMember` is an instance variable declared by the *member class*, not if `dataMember` belongs to the enclosing class.
  - `x = EnclosingClass.this.dataMember` allows access to `dataMember` that belongs to the enclosing class.
- Inner classes can be nested to any depth and the `this` mechanism can be used with nesting.

# this and Member Classes

```
public class EnclosingClass {
    private int i,j;

    private class MemberClass {
        private int i; public int j;

        public void memMethod( int i ) {
            int a = i;           // Assign param to a
            int b = this.i;      // Assign member's i to b
            int c = EnclosingClass.this.i; // Assign top-level's i to c
            int d = j;           // Assign member's j to d
        } }

    public void ecMethod() {
        MemberClass mem = new MemberClass();
        mem.memMethod( 10 );
        System.out.println( mem.i + mem.j ); // is this a bug?
    }

    public static void main(String args[]) {
        EnclosingClass e = new EnclosingClass();
        e.ecMethod(); }
}
```

**>java EnclosingClass**  
**0**  
**>**

# new Revisited

- Member class objects can only be created if they have access to an enclosing class object.
- This happens by default if the member class object is created by an instance method belonging to its enclosing class.
- Otherwise it is possible to specify an enclosing class object using the `new` operator as follows
  - `MemberClass b =  
    anEnclosingClass.new MemberClass();`

# Local Classes

- A local class is a class declared within the scope of a compound statement, like a local variable.
- A local class is a member class, but cannot include static variables, methods or classes. Additionally they cannot be declared `public`, `protected`, `private` or `static`.
- A local class has the ability to access *final* variables and parameters in the enclosing scope.

# Local Class Example

```
public class EnclosingClass {
    String name = "Local class example";

    public void ecMethod( final int h, int w ) {
        int j = 20; final int k = 30;

        class LocalClass {
            public void aMethod() {
                System.out.println( h );
                // System.out.println( w ); ERROR w is not final
                // System.out.println( j ); ERROR j is not final
                System.out.println( k );
                // System.out.println( i ); ERROR i is not declared yet
                System.out.println( name); // normal member access
            }
        }

        LocalClass l = new LocalClass(); l.aMethod();
        final int i = 10; }

    public static void main(String args[] ) {
        EnclosingClass c = new EnclosingClass(); c.ecMethod( 10, 50 ); }}

```

>Java EnclosingClass  
10  
30  
Local class example  
>

# Anonymous Classes

- An anonymous class is a local class that does not have a name.
- An anonymous class allows an object to be created using an expression that combines object creation with the declaration of the class.
- This avoids naming a class, at the cost of only ever being able to create one instance of that anonymous class.
- This is handy in the AWT and thread creation.

# Anonymous Class Syntax

- An anonymous class is defined as part of a `new` expression and *must* be a subclass or implement an interface

```
new className( argumentList ) { classBody }  
new interfaceName() { classBody }
```

- The class body can define methods but cannot define any constructors.
- The restrictions imposed on local classes also apply

# Using Anonymous Classes

```
import java.awt.*; import java.awt.event.*;import javax.swing.*;

public class MainProg {
    JFrame win;

    public MainProg( String title ) {
        win = new JFrame( title );

        win.addWindowListener(
            new WindowAdapter() {
                public void windowClosing( WindowEvent e ) {
                    System.exit( 0 );
                }
            });
    }

    public static void main( String args[] ) {
        MainProg x = new MainProg( "Simple Example" );
    }
}
```

# Active Object Pattern

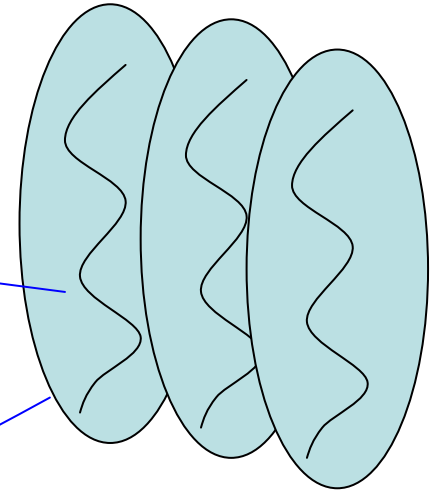
Decouple *method execution* from  
*method invocation* to enhance  
concurrency

呼叫 <> 立即執行

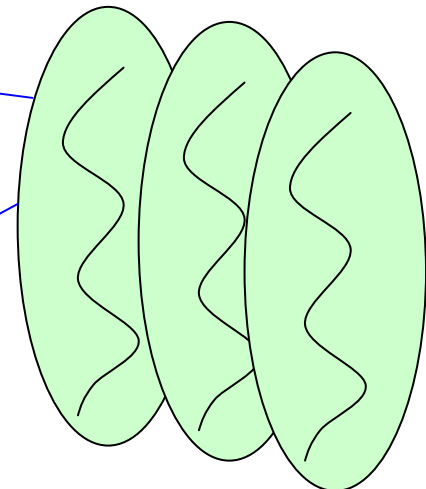
# Passive Objects

```
class Buffer { // no threads; run in client's threads
  private int count = 0;
  private Object buffer;
  public synchronized void put(Object x) {
    while (count == 1) wait();
    buffer=x; count=1;
    notify();
  }
  public synchronized Object get() {
    Object x;
    while (count == 0) wait();
    count=0;
    notify(); // change to notifyAll();
    return x;
  }
}
```

**Producer Threads**

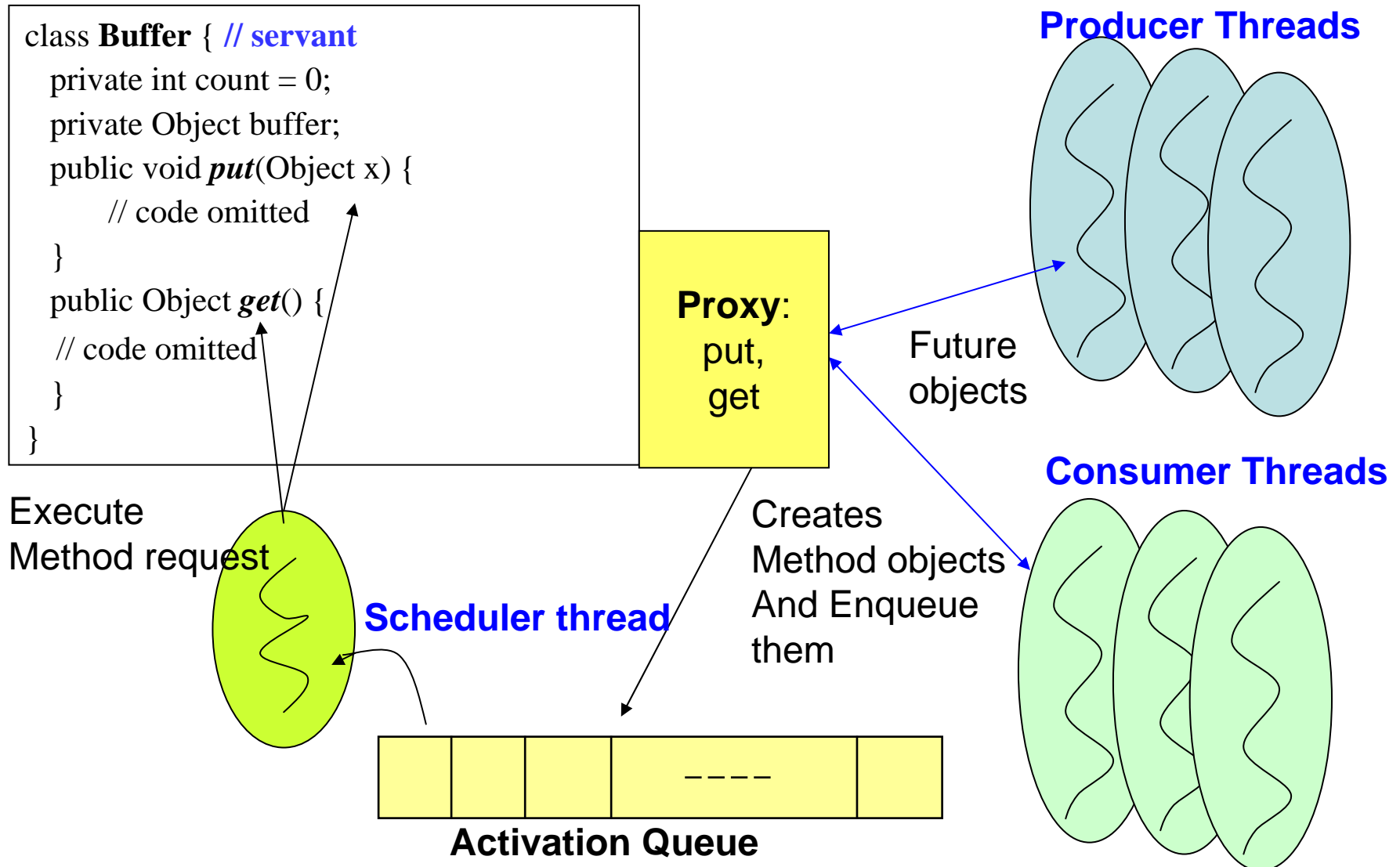


**Consumer Threads**



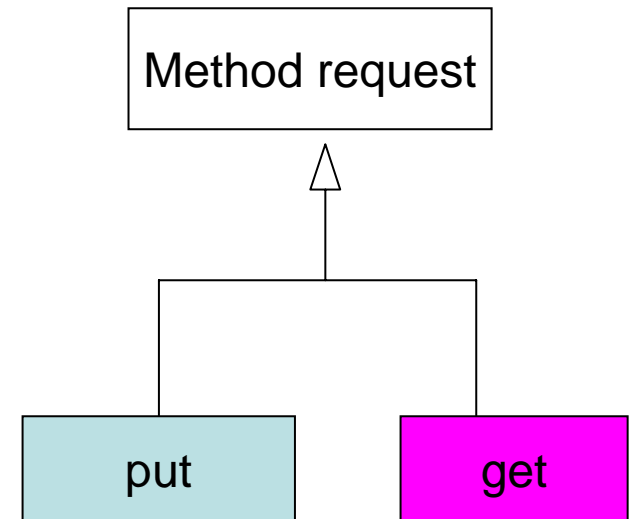
# Active Objects: Basic Idea

**Decouples method execution from method invocation**



# Participants of Active Objects

- Client (threads)
- Proxy (of servant)
- Servant (Buffer)
- Scheduler
  - Activation queue
  - Dispatcher
- Method request objects
- Future (results of method call)



# Proxy

Only one scheduler (consumer) threads

```
class Buffer { //sequential
  private int count = 0;
  private Object buffer;
  public void put(Object x) {
    // code omitted
  }
  public Object get() {
    // code omitted
  }
}
```

```
Class Proxy { // not synchronized

  public void put(Object x) {
    create a Put request object;
    enqueue it to Activation Queue
  }
  public Future get() {
    create a Get request object;
    enqueue it to Activation Queue
  }
  public Proxy(Servant b, Scheduler s)
  { ... }
  protected Servant buffer;
  protected Scheduler scheduler;
}
```

```
MethodRequest m = new Put(buffer, x);
Scheduler.enqueue(m);
```

# Method Request Objects

```
Class MethodRequest {  
    public abstract Boolean guard();  
    public abstract void execute();  
}
```

```
Class Put extends MethodRequest {  
    protected Servant buffer;  
    protected Object message;  
    public Boolean guard() {  
        return !buffer.isFull();  
    }  
    public void execute() {  
        buffer.put(message);  
    }  
    public Put(Servant b, Object m) {  
        buffer = b; message = m;  
    }  
}
```


```
Class Get extends MethodRequest {  
    protected Servant buffer;  
    protected Future result;  
    public Boolean guard() {  
        return !buffer.isEmpty();  
    }  
    public void execute() {  
        result.setMessage(buffer.get());  
    }  
    public Get(Servant b, Future f) {  
        buffer = b; result = f;  
    }  
}
```

# Scheduler (A Separate Thread)

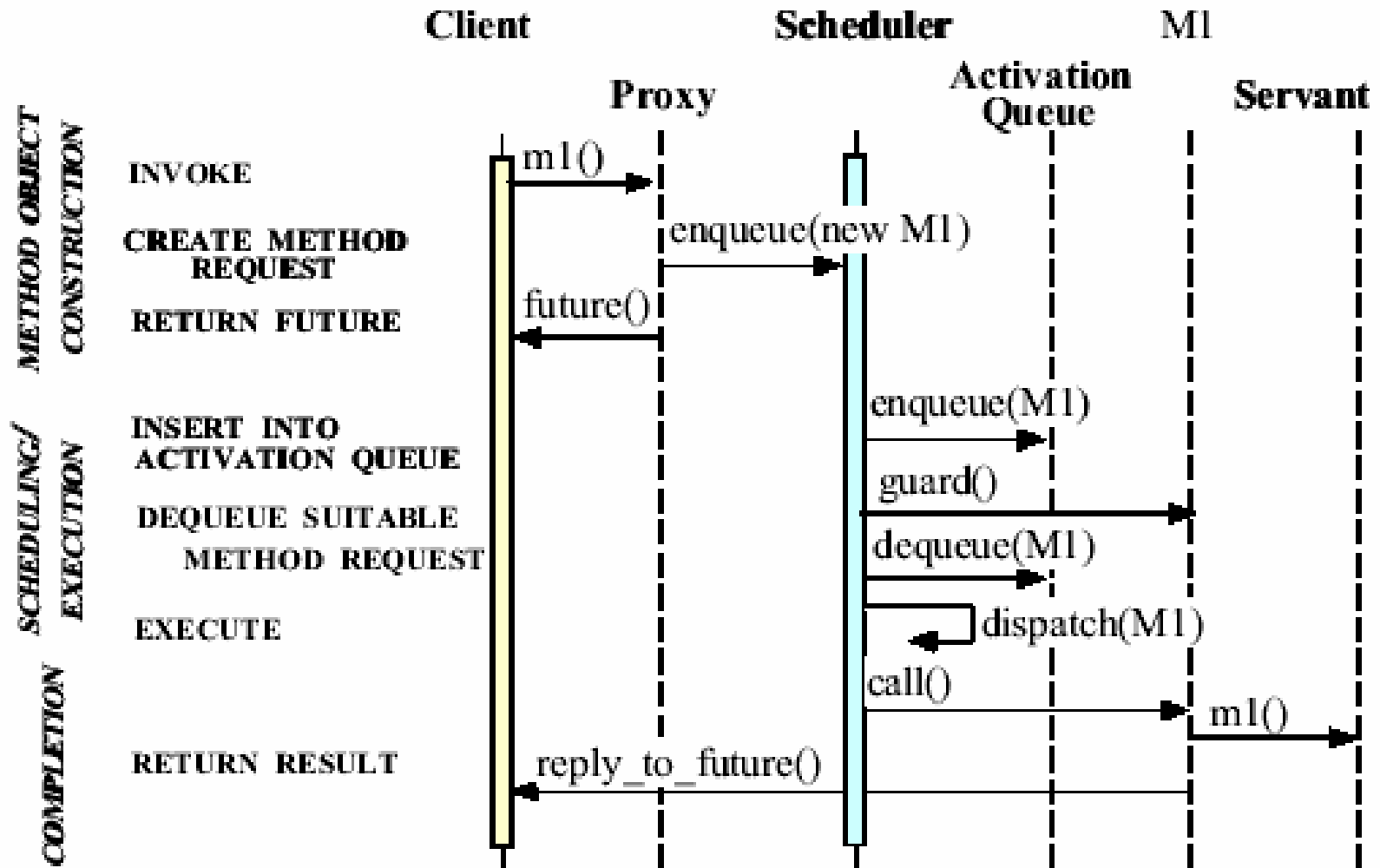
- A scheduler maintain an *Activation Queue* and execute pending *MethodRequests* whose synchronization constraints are met.

```
Class Scheduler implement Runnable {  
    protected ActivationQueue aq;  
    protected static Thread  
    public enqueue(MethodRequest m) {  
        aq.enqueue(m);  
    }  
    public void run() { dispatch(); }  
    public void dispatch() {  
        ...  
    }  
    public Scheduler(ActivationQueue,aq) {  
        this.aq = aq;  
        Thread t = new Thread (this);  
    }  
}
```

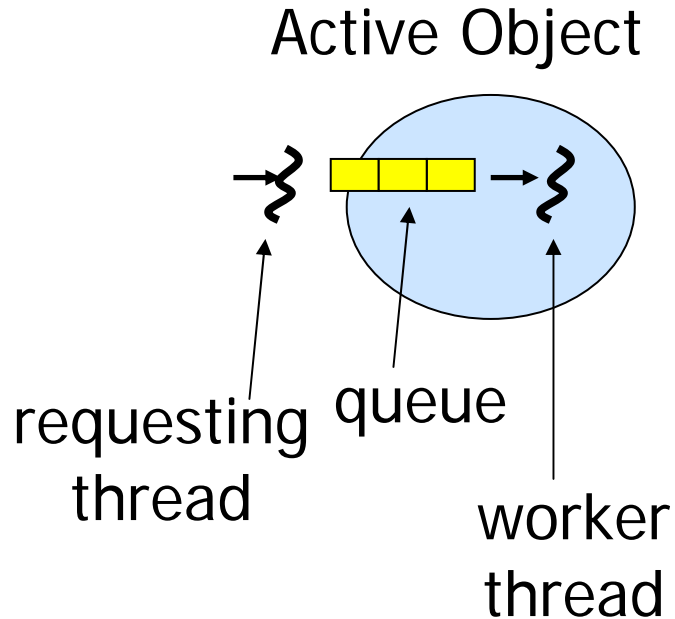
```
Iterator i = aq.iterator();  
While (i.more()) {  
    m = i.next();  
    if (m.guard()) {  
        aq.deueue(m);  
        m.execute();  
    }  
}
```



# Collaborations



# Summary: Active Object



- Decouples
  - Method execution from invocation
- Helps prevent
  - Blocking
  - Deadlock
- Can improve performance
  - Through additional concurrency
- Drawbacks
  - Queuing overhead
  - Context-switch overhead
  - Implementation complexity

# Thread Cancellation

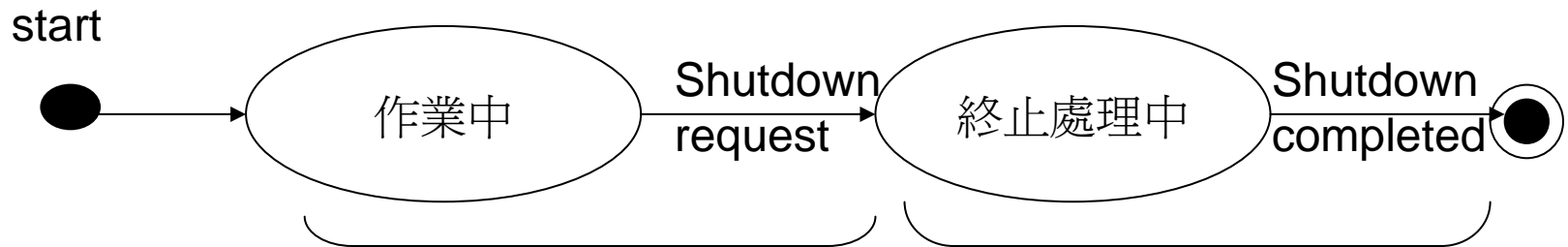
```
public class InterruptibleThread implements Runnable
{
    public void run() {
        while (true) {
            /**
             * do some work for awhile
             */

            if (Thread.currentThread().isInterrupted()) {
                System.out.println("I'm interrupted!");
                break;
            }
        }

        // clean up and terminate
    }
}
```

# Two-Phase Termination Pattern

- 優雅的終止 執行緒
- Provide for the *orderly shutdown* of a thread or process through the setting of a *latch*.
- The thread or process checks the value of the latch at strategic points in its execution
  - Flag variable: *isShutdown*



```
public class Main {
    public static void main(String[] args) {
        System.out.println("main: BEGIN");
        try {
            // 啓動執行緒
            CountupThread t = new CountupThread();
            t.start();

            // 稍微空出一段時間
            Thread.sleep(10000);

            // 對執行緒送出終止請求
            System.out.println("main: shutdownRequest");
            t.shutdownRequest();

            System.out.println("main: join");

            // 等待執行緒結束
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("main: END");
    }
}
```

```

public class CountupThread extends Thread {
    private long counter = 0;
    // 已經送出終止請求則為true
    private volatile boolean shutdownRequested = false;
    // 終止請求
    public void shutdownRequest() {
        shutdownRequested = true;
        interrupt();
    }
    // 判斷終止請求是否已經送出
    public boolean isShutdownRequested() { return shutdownRequested; }
    // 動作
    public final void run() {
        try {
            while ( !shutdownRequested ) { doWork(); }
        } catch (InterruptedException e) {
        } finally { doShutdown(); }
    }
    // 作業
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }
    // 終止處理
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}

```

# Thread-Safety

- Local variables are automatically thread-safe
- Instance variables of an object whose methods are run by multiple threads are not safe
  - declare methods that access it `synchronized`
  - or declare the variable `volatile`
- Class (a.k.a. static) variables are not thread-safe
  - declare methods that access it `synchronized`
  - or declare the variable `volatile`
  - or make them of type `ThreadLocal` or `InheritableThreadLocal` (see next slide)

# Volatile

- Consider the following code in thread's run()

```
obj.currentValue=5;
for ( ;; ) {
    System.out.print(obj.currentValue+" , " );
    Thread.sleep(1000);
}
```

- Another thread concurrently *increases* the value of `obj.currentValue`
- Surprisingly, the thread will print

5,5,5,5,5,...

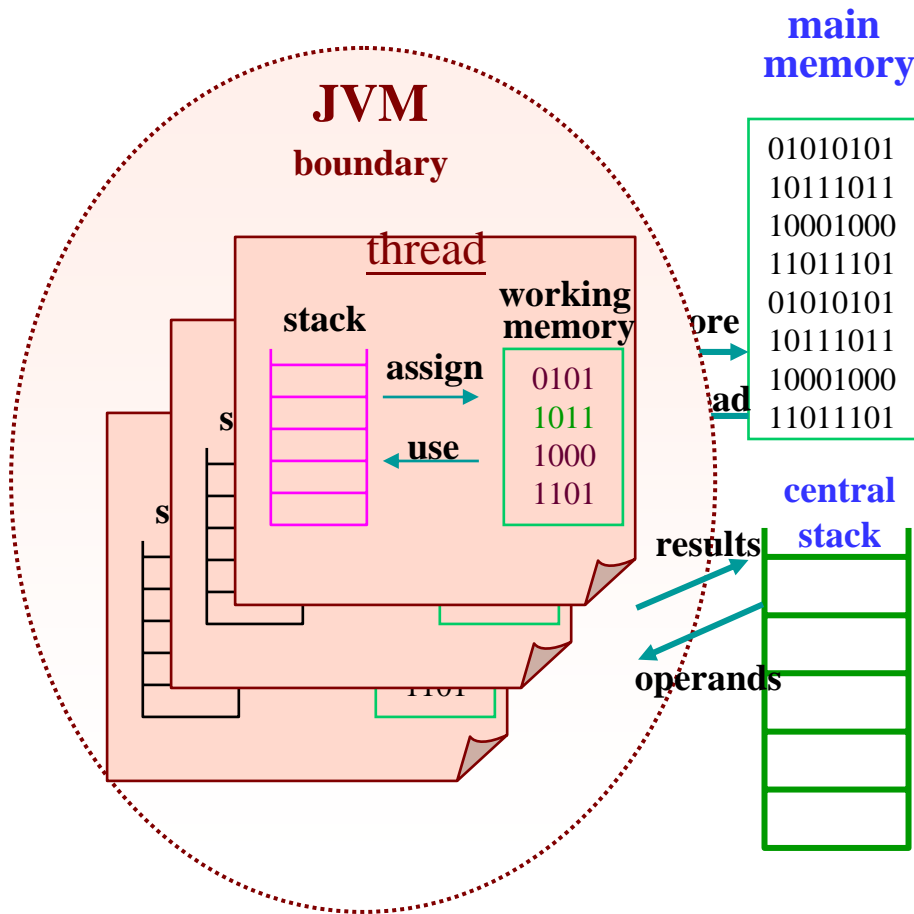
# Volatile

- Since the value of `obj.currentValue` *is not modified in the loop*, the compiler simply continues printing the same value
- Define the variable as *volatile* by:  

```
volatile int currentValue;
```
- This will cause to perform *actual memory access and value update* for every access to the variable

# Java's memory model

Working memory  $\leftrightarrow$  main memory: synchronization issues



- **weak consistency:** shared variables updates are only made visible to other threads in synchronised code blocks
- uses a computation model based on a (global) stack
- Efficient code interpretation but not execution on register based processors

Java memory model

# Thread-Specific Storage

- 每個thread自己的保管箱
- Allows each thread to have *its own copy of data*, called ***thread-specific data***
- Useful when you do not have control over the thread creation process (i.e., when using a *thread pool*)

# Logging Example without Using TSS

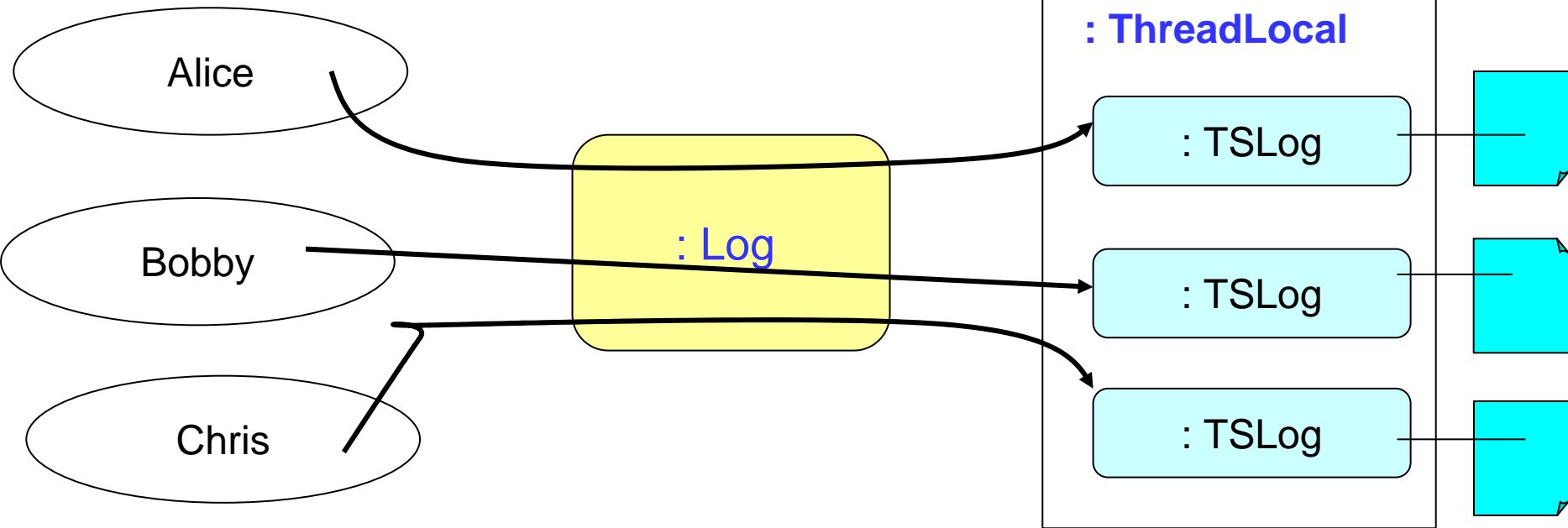
```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("BEGIN");  
        for (int i = 0; i < 10; i++) {  
            Log.println("main: i = " + i);  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e)  
            { }  
        } // for  
        Log.close();  
        System.out.println("END");  
    }  
}
```

All logged messages written to  
The same log file.

```
import java.io.PrintWriter;  
import java.io.FileWriter;  
import java.io.IOException;  
public class Log {  
    private static PrintWriter writer = null;  
    // 初始化writer欄位  
    static {  
        try {  
            writer = new PrintWriter(new FileWriter("log.txt"));  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    // 加入一筆log  
    public static void println(String s) {  
        writer.println(s);  
    }  
    // 關閉log  
    public static void close() {  
        writer.println("==== End of log =====");  
        writer.close();  
    }  
}
```

# One Log File for Each Client Thread

Client threads



# Thread-Specific Storage

- The Thread-Specific Storage design pattern allows multiple threads to use one '*logically global*' access point to retrieve an object that is local to a thread, without incurring locking overhead for each access to the object
- Thread-external data
- Class *java.lang.ThreadLocal* (like a hashtable)

```
ThreadLocal t1 = new ThreadLocal();  
MyObj mo = t1.get(); // thread id is the key  
if (mo= null) t1.set( new MyObj() );
```

- Each thread will get its own *MyObject*.

# Modified Example

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName() + " BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        Log.close();
        System.out.println(getName() + " END");
    }
}
```

The same code in the  
Previous Main class.



```
public class Log { // proxy object
    private static final ThreadLocal tsLogCollection
        = new ThreadLocal();

    // 加入一筆log
    public static void println(String s) {
        getTSLog().println(s);
    }

    // 關閉log
    public static void close() {
        getTSLog().close(); // forwarding
    }

    // 取得執行緒特有的log
    private static TSLog getTSLog() {
        TSLog tsLog = (TSLog)tsLogCollection.get();

        // 如果執行緒是第一次呼叫，就建立新檔案並登錄log
        if (tsLog == null) {
            tsLog = new
                TSLog(Thread.currentThread()
                    .getName() + "-log.txt");
            tsLogCollection.set(tsLog); // forwarding
        }
        return tsLog;
    }
}
```

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;
```

```
public class TSLog { // use thread-specific data
    private PrintWriter writer = null;
    // 初始化writer欄位
    public TSLog(String filename) {
        try {
            writer = new PrintWriter(new FileWriter(filename));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // 加入一筆log
    public void println(String s) {
        writer.println(s);
    }
    // 關閉log
    public void close() {
        writer.println("==== End of log ====");
        writer.close();
    }
}
```

Client thread → Log → TSLog

```
public class Main {
    public static void main(String[] args) {
        new ClientThread("Alice").start();
        new ClientThread("Bobby").start();
        new ClientThread("Chris").start();
    }
}
```