Background Concepts for Distributed Computing: Hardware, OS, Design Patterns

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)

CPU, Device, Bus

Computer Bus	Allows CPUs and devices to communicate and access memory
CPU	Interprets general-purpose commands in instruction registers that ask it to perform arithmetic on data registers, interact with devices and memory.
Device	Performs special-purpose computations and also has registers, accessible by the CPUs through the bus. Example: real-time clock, keyboard, display, network.
Device Readiness	Device ready to receive (load) data or send (save) data to CPU indicated in a device register bit, signaling non-full or nonempty register, respectively

Program, Threads, Processes

Program	Procedures (with local variables) + global variables with a distinguished main procedure
Thread	Execution of a method call chain starting at some distinguished method called the thread root. Has its own stack. Concurrent or interleaved execution of threads possible and threads can start children threads and terminate before their descendants.
Main Thread	Thread that executes the main method – started when the program is executed
Process	Represents an executing program = address space + main and descendent threads
Address space	A portion of memory that stores the (code) instructions, global variables and stacks of an executing program
Atomic execution	Access to a critical region in shared memory by a thread that is uninterrupted by another thread – maintains an invariant. Supported by thread coordination schemes .

OS, Kernel, Kernel Mode

OS	Creates processes and their threads, schedules them on CPUs, accesses device registers, supports IPC, provides thread coordination – beautifies hardware and mediates resources
Kernel	Portion of OS that executes in kernel mode.
Kernel Mode	A hardware mode with special privileges such as accessing of device registers, all of memory, ….
Use Mode	Hardware mode in which access is limited to a processes' address space
Trap instruction (software interrupt)	Instruction taking an index argument executed in user mode that executes indexed trap handler in kernel mode, which can execute return from trap to return to user mode. Trap handler can execute in special kernel thread or as part of trapping user thread. Trap handler = system call + error handler
System call	Trap handler that services a system call such as get or put

Context Block, Context Switch, Process Switch

Thread Context Block	Area in kernel memory representing a thread (stack pointer, program counter, other registers)
Process Context Block	Portion of OS represents a process – address space, threads
Thread switch	Giving a CPU to another thread of same process
Process switch	Giving a CPU to another thread of different process
Current, ready, blocked thread	Thread being executed by CPU, ready to execute on a CPU, not ready to execute. Queues for non current threads.
Context Switch	Switching from user to kernel model (to say execute system call) and often results in thread switch

Interrupts

Enabling/disabling device interrupt posting	Instruction asking a device to post/not post a signal on the bus indicating it is ready. Prevents polling of ready bit.
Interrupt servicing	Execution in kernel mode of a handler by the CPU that services the interrupt. Handler indexed by interrupting device.
	Executed by current user thread on a CPU or special kernel thread. Servicing of clock interrupts results in rescheduling, Servicing of interrupts of I/O devices results in I/O.
	Interrupts can be directed to a special CPU, designated CPU, or any CPU, with only one of them servicing it.
Enabling/Disabling CPU interrupts servicing	Instruction executed in kernel mode to enable/disable interrupt servicing by a CPU. Interrupt routines (on single CPU) disable interrupts to ensure atomic access (by that CPU) to shared kernel data structures such as context blocks.

Queues, Bounded Buffer

Bounded Queue	FIFO, with small or large bound, into which items can be produced (enqueued) and from which items can be consumed (dequeued). Full queue during enqueuing and empty queue on dequeuing is an error or noop if accessed by a single thread.
Synchronized Bounded Buffer	Queue accessed by multiple threads. Full queue during enqueuing and empty queue on dequeuing results in blocking.
Half-Synchronized Bounded Buffer	Queue accessed by multiple threads, some of which cannot block. Full queue during enqueuing and empty queue on dequeuing results in blocking for some threads and errors or no-ops for others

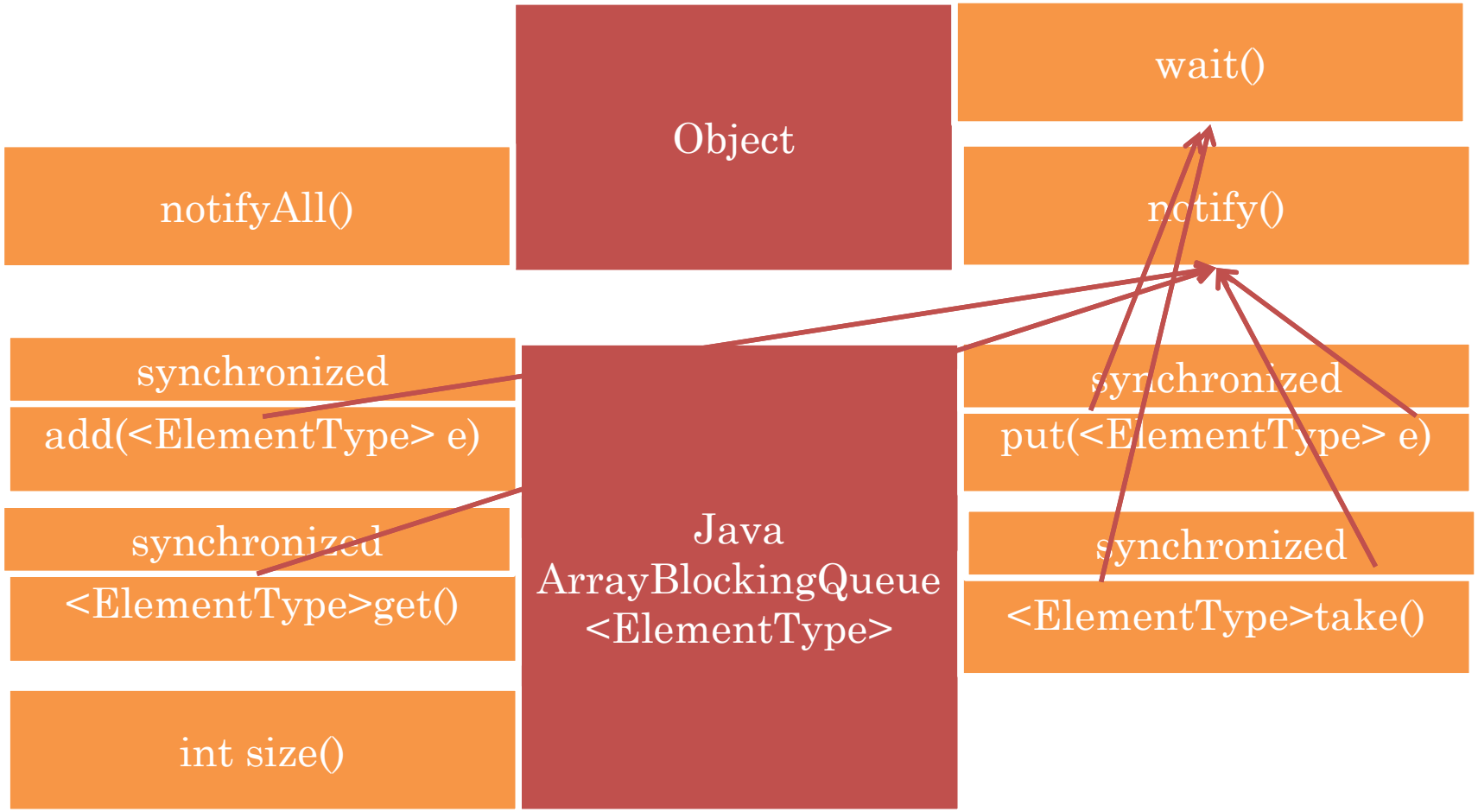
Observer Pattern

Observable/Observer	Observer objects register themselves with observables and execute notification code when interesting events occur in the observables.
	Enabling interrupt posts corresponds to registries. Execution of error trap and interrupt handlers corresponds to execution of notification methods, except only one interrupt routine is executed. Registration and notification threads can be different.
Callback	An observable notification method called by an observer method in a lower layer

Queues, Bounded Buffer (Overview)

Bounded Queue	FIFO, with small or large bound, into which items can be produced (enqueued) and from which items can be consumed (dequeued). Full queue during enqueuing and empty queue on dequeuing is an error or noop if accessed by a single thread.
Synchronized Bounded Buffer	Queue accessed by multiple threads. Full queue during enqueuing and empty queue on dequeuing results in blocking.
Half-Synchronized Bounded Buffer	Queue accessed by multiple threads, some of which cannot block,
	Full queue during enqueuing and empty queue on dequeuing results in blocking for some threads and errors or no-ops for others
	Threads executing interrupt routines cannot block: that would unnecessarily delay the threads if they are user-level, delay all interrupts processing, result in lost interrupts (interrupt routines consume data produced by agents they cannot control) lead to deadlocks if condition for unblocking depends on a future interrupt processing

Half Synchronized Bounded Buffer



Not synchronizing

Synchronizing

Blocking Implemented how?

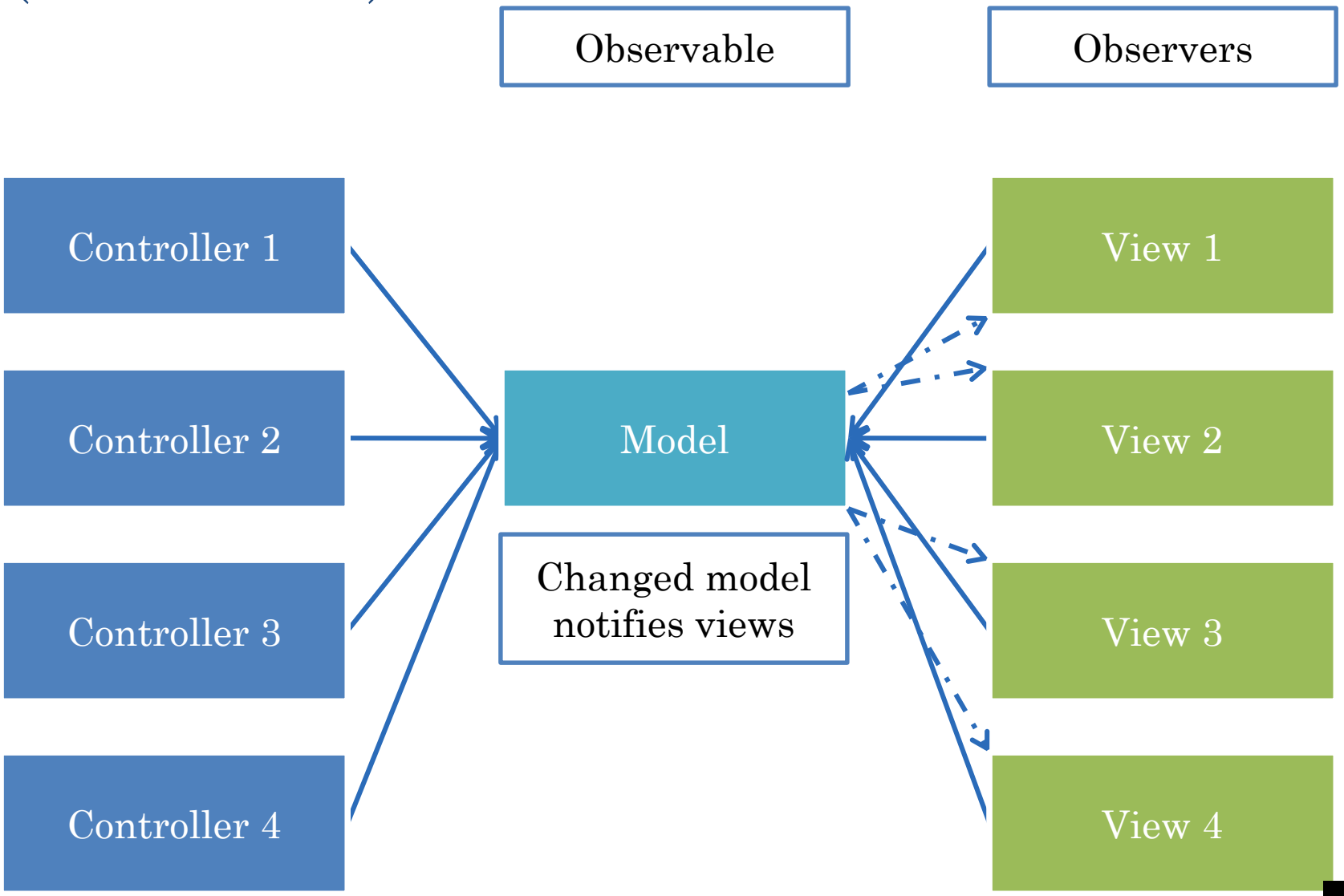
Wait, Notify, Synchronized

A single synchronized method can be executed on an instance at any time.
Different synchronized methods can execute on different instances at the same time.
Wait blocks the thread that executes it (usually after checking some condition)
Notify/notifyAll unblocks some/all queued thread (also usually executed after checking the condition(s) on which wait is blocked)
Wait, notify(), notifyAll() share access to the thread queue in which waiting threads are blocked and hence should be executed by synchronized methods of the same instance
Wait and notify provide finer-grained synchronization than a kernel method with interrupt processing disabled.

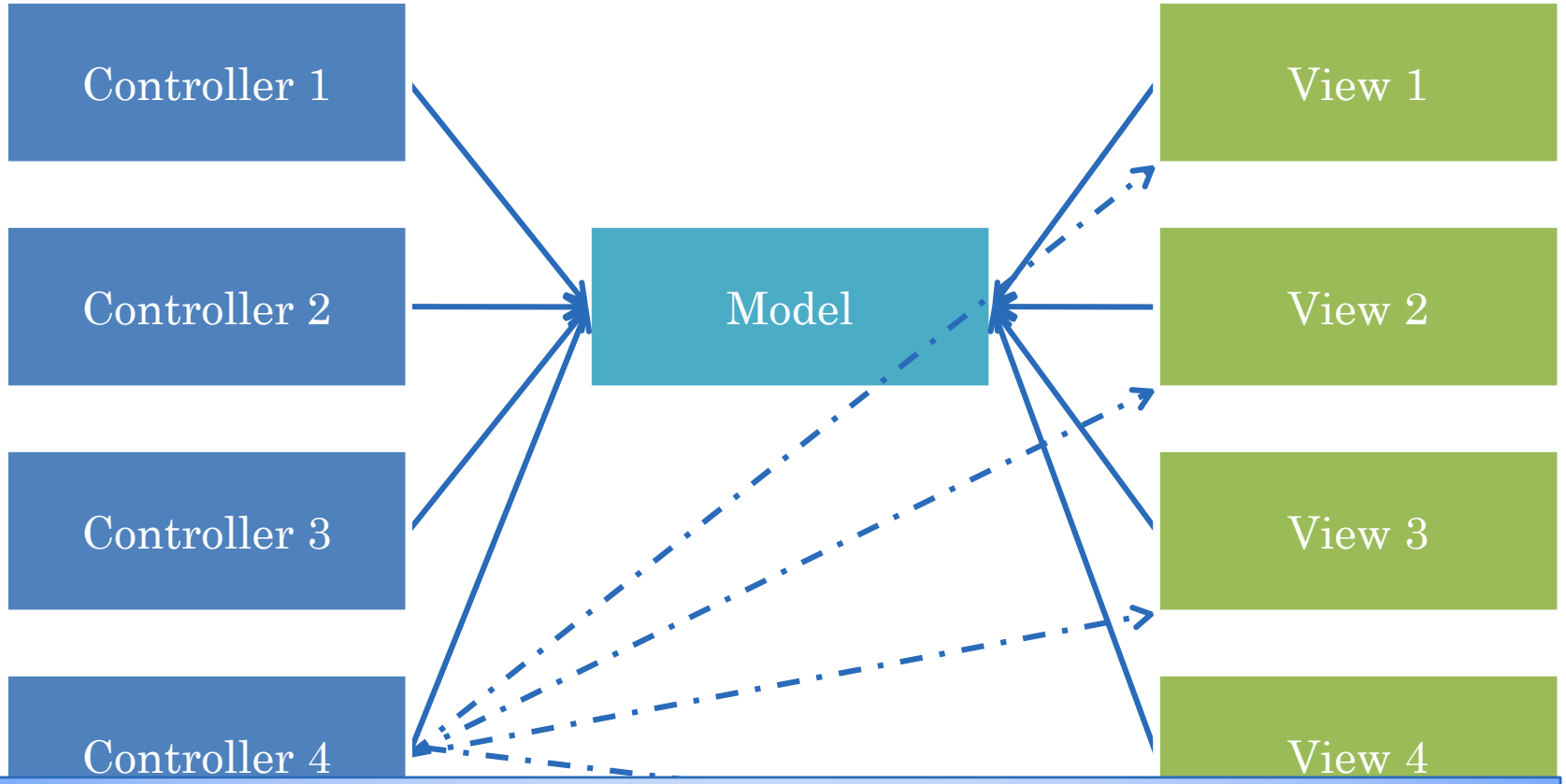
Observer vs. Notify/Wait

Observable/Observer	Observer objects register themselves with observables and execute notification code in same thread when interesting events occur in the observables.
	Enabling interrupt posts and waiting corresponds to observer registration.
	Execution of error trap and interrupt handlers and unblocking of wait corresponds to execution of notification methods.
	In trap handlers and observer notification methods only one thread is involved, while wait/notfy() and interrupt handlers support inter-thread coordination and unblocking.
	In trap handlers and interrupt handlers, an event of interest results in a a single reaction, while notification methods and notifyall can result in multiple reactions.

(Smalltalk) MVC and Observer Pattern



A Tale of Two MVCs: Web MVC



In Http-based "MVC" a single view and controller exist in the browser and the model in the server. A Model cannot initiate actions in the browser so the controller directly communicates with the view

Example MVC Trace

I***{main}(AddedPropertyChangeListener) EvtSrc(AMeaningOfLifeModel) examples.nio.manager.client.AMeaningOfLifeClientSender@2dda6444

I***{main}(AddedPropertyChangeListener) EvtSrc(AMeaningOfLifeModel) examples.nio.manager.mvc.AMeaningOfLifeView@49c2faae

42

I***{main}(SetProperty) EvtSrc(AMeaningOfLifeModel) Meaning = 42

I***{main}(NotifiedPropertyChangeEvent) EvtSrc(AMeaningOfLifeModel) java.beans.PropertyChangeEvent[propertyName=Meaning; oldValue=; newValue=42; propagationId=null; source=examples.nio.manager.mvc.AMeaningOfLifeModel@17f052a3]->[examples.nio.manager.client.AMeaningOfLifeClientSender@2dda6444, examples.nio.manager.mvc.AMeaningOfLifeView@49c2faae]

Meaning of life:42

{<thread name>} (<Event Type>) EvtSrc (<Class of code announcing event>) <Event-Specific Parameters>

Observer vs. Callback

Observable/Observer	Observer objects register themselves with observables and execute notification code in same thread when interesting events occur in the observables.
Callback	Notification method called on a higher level layer by an observer method in a lower layer

Command vs. Action Object

Observer/callback object

execute (params)

Commmand (method + parameters)

Constructor (targetObject, params)

execute ()

Provides an execute operation to perform some action.

Constructor can take target object and parameters of operation as arguments.

Action vs. command object ←→ put vs put 'a'

A callback is an operation (method) that can be invoked on many different arguments decided by the calling service

Use of command object?

A command is an operation (method) invocation whose parameters are decided by the command or its creator and not the calling service

Command vs. Action Object

Observer/callback object

execute (params)

Commmand (method + parameters)

Constructor (targetObject, params)

execute ()

Provides an execute operation to perform some action.

Constructor can take target object and parameters of operation as arguments.

Action vs. command object ←→ put vs put 'a'

A callback is an operation (method) that can be invoked on many different arguments decided by the calling service

Use of command object?

A command is an operation (method) invocation whose parameters are decided by the command or its creator and not the calling service

Command vs. Action Object (Review)

Observer/callback object

execute (params)

Commmand (method + parameters)

Constructor (targetObject, params)

execute ()

Provides an execute operation to perform some action.

Constructor can take target object and parameters of operation as arguments.

Action vs. command object ←→ put vs put 'a'

A callback is an operation (method) that can be invoked on many different arguments decided by the calling service

Use of command object?

A command is an operation (method) invocation whose parameters are decided by the command or its creator and not the calling service

Async Command Execution in New Thread

```
public class APutCommand implements Runnable {
   char addition;
   public APutCommand (char anAddition) {
            addition = anAddition;
   }
   @Override
   public void run() {
      SomeBufferFactory.getSingleton().put(addition);
   }
}
```

Execute operation call in some other but not new thread?

```
Runnable aPutCommand = new APutCommand('a');
```

```
Thread aThread = new Thread(aPutCommand);
```

```
aThread.start();
```

```
aThread.join();
```

Async Command Execution in Thread Pool

```
public class APutCommand implements Runnable {
   char addition;
   public APutCommand (char anAddition) {
            addition = anAddition;
   }
   @Override
   public void run() {
      SomeBufferFactory.getSingleton().put(addition);
   }
}
```

Retrieve returned value after wait

```
Runnable aPutCommand = new APutCommand('a');
```

```
Future aFuturePutCommand =
MyExecutorServiceFactory.getSingleton().submit(aPutCommand);
```

```
aFuturePutCommand.get();
```

Async Command Execution With Return Value

```
public class AGetCommand implements Callable<Character>{
    @Override
    public Character call() {
        return SomeBufferFactory.getSingleton().get();
    }
}
```

```
Callable aGetCommand = new AGetCommand();
```

```
Future aFutureGetCommand =
MyExecutorServiceFactory.getSingleton().submit(aGetCommand);
```

```
char result= aFutureGetCommand.get();
```

Parsing interpreted line?

Interpreter Command Execution

```
public class APlusExpression implements Callable<Callable>{
   Callable leftOperand, rightOperand;
   public APlusExpression (Callable aLeftOperand,  Callable aRightOperand) {
            leftOperand = aLeftOperand;
            rightOperand = aRightOperand;
   }
   @Override
   public Callable call() {
      return leftOperand.call() + rightOperand.call();
   }
}
```

```
Callable anExpression = parser.parseNextLine()
```

```
System.out.println("Evaluated expression:" + anExpression.call());
```

Design pattern used besides command object?

Composite Pattern: Parent node implements the same interface as its children (like visitor pattern)

Undo?

Undo/redo

```
public class AnUndoableGetCommand implements Runnable {
   char deletion;
public void execute() {
       deletion = SomeBufferFactory.getSingleton().get();
   }
  public void undo() {
       SomeBufferFactory.getSingleton().put(deletion);
  }
}
```

```
Undoable anUndoableGetCommand= new AnUndoableGetCommand ();
```

```
UndoerFactory.getSingleton().submit(anUndoableGetCommand);
```

```
UndoerFactory.getSingleton().undo();
```

```
UndoerFactory.getSingleton().redo();
```

Command Object

An object that represents a command =operation + args
Provides a parameterless “execute” method to invoke the command()
Can have a constructor to take invocation parameters
A command submitter wishes invocation of the method in the future by a command executor service, which is oblivious of what it does
May provide additional methods such as undo.

Command Object Services

A thread object starts a new thread to execute commands at submission time.

A thread-pool executor service puts commands in bounded buffer consumed by one or more existing threads that execute consumed items

An undoer services puts command in a history and allows their undo and redos.

The parser of a command line creates a hierarchy of command objects and interprets it by calling the execute() method of the root of the tree.

Factories and Singletons?

```
public class AnUndoableGetCommand implements Runnable {
   char deletion;
public void execute() {
      deletion = SomeBufferFactory.getSingleton().get();
   }
  public void undo() {
      SomeBufferFactory.getSingleton().put(deletion);
  }
}
```

```
Undoable anUndoableGetCommand= new AnUndoableGetCommand ();
```

```
UndoerFactory.getSingleton().submit(anUndoableGetCommand);
```

```
UndoerFactory.getSingleton().undo();
```

```
UndoerFactory.getSingleton().redo();
```

Singletons and Factories

Factory Class/Object for class C	Provides a static or instance method to return an object of some class bound (after possibly instantiating it) to the factory
	Used for singletons and selecting some implementation of an interface
Factory selector or Factory-factory or abstract factory	A factory can be retrieved from a factory-factory to choose an implementation of an interface
Singleton class	A class with only one instance
	A factory for that class would always return that instance and can be used to enforce singletons.
	A factory class with static methods can be replaced with an instance of a singleton factory.

Summary

Single-thread concepts: Relationship between observer pattern, callbacks, upcalls,

Multi-thread concepts: Extend single-thread coordination, and will be extended to multi-process coordination

Relationship between interrupts, device registers, producer-consumer problem, blocking queue, synchronized methods, wait, notify, I/O, disabling of interrupt processing, disabling/enabling interrupt posting

Command objects – method calls to be executed by some general service in same or different (possibly new) thread

Factories and singletons