CO3401 Advanced Software Engineering Techniques

Tutorial Exercise Book 1: Concurrency ***with answers***

Introduction to Concurrency

1. Explain the following terms: multi-programming, multi-processing, parallel programming, concurrency, multi-threading.

   Several programs running at the same time (simultaneously). This could be true parallel programming with multiple instructions executing at the same instant on several processors or simply switching between each program so over a period of time each program progresses.

   Several parts of the same program running at the same time

   Multiple processors executing code at the same instant

   Multiple programs or program parts executing at the same time, possibly by swapping among them.

   Several parts of a program executing at the same time, where those parts can share variables.

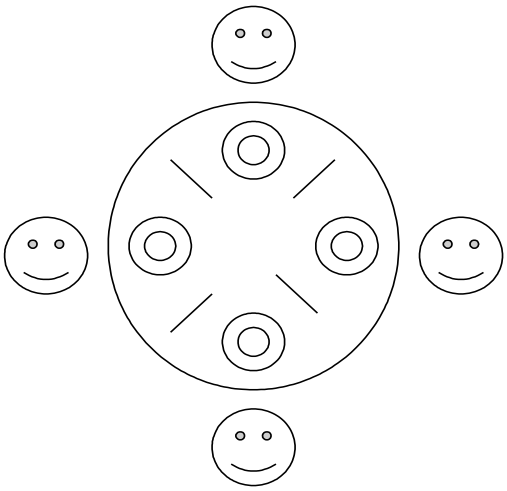
2. Deadlock occurs when several processes compete for several resources and each process holds a resource and requires another resource that is held by one of the other processes. There is a cycle of processes each unable to proceed without a resource held by the neighbour.
   a. Describe a real world situation where deadlock could occur.
   b. Define a simple rule for allocating resources that guarantees that deadlock can't occur. Are there any disadvantages to this rule?

   I get the data projector and take it to a class room. My colleague gets a screen from another store and takes it to a different class room. We both go to the relevant storeroom to collect the other item and wait.

   Number the resources and insist that resources must be taken in order. E.g. if the screen had a lower number than the projector, both of us would have to get that first, the one who did would then be able to get the projector. This approach stops there being a cycle of threads waiting for resources.

   The disadvantage is that if I'm going to use the projector for some time before I use the screen, I have to hang on to the screen even though I'm not using it because I have to collect it first. So the method may be less efficient.

3. A well-known problem used to illustrate the deadlock problem is known as the “dining philosophers”. A group of philosophers are sitting around a round table with an individual bowl of food in front of each philosopher. Between each bowl is a chopstick. They are only allowed to eat if they have picked up the two chopsticks on the left and right side of their bowl. Each philosopher alternates between eating and thinking for random periods.



   a. Why are such standard problems used in discussing concurrency? (Hint: model)

They provide a simple, clear model of a common problem. A solution to such a problem can often be adapted to other similar situations.

   b. Explain how deadlock could occur.

Each philosopher picks up the chopstick to his left and then waits for the stick to his right to come free.

   c. If your rule from 2 was applied to the situation, could deadlock occur?

No, because each stick would be numbered (e.g. 1-4) It is not possible for each philosopher to pick up the fork to his left first. The last one has to pick up the one to the right. This means it is impossible for a circle of waiting philosophers to be set up and so at least one can always proceed – no deadlock.

   d. Starvation is when a process is unable to get the resources it requires because through bad luck, every time it is in a race with another process to get a resource or requests for resources are processed in such a way it is never lucky enough to get the resources it requires. If your rule was applied to the dining philosophers, can you design a sequence of requests that end up with a philosopher starving?

If philosophers are allowed to wait for a fork until it comes free and no philosopher eats forever, then sooner or later everyone must be able to proceed. If they repeatedly ask for a fork rather than wait for it, or if waiting threads are not released in order, it may be possible for starvation to occur.

4. A waiter writes orders for food on a blackboard that can hold up to N orders. A member of the kitchen staff copies orders from the blackboard and takes a number of them to the kitchen. Identify the potential concurrency problems that could arise.

How does the waiter know if the staff has taken an order through to the kitchen so that the slot can be reused. How does the kitchen staff know if an order that has been taken through hasn’t been replaced by an identical order.

5. A thread is putting readings into a buffer (based on an array of size N). Another thread is removing the readings. Identify the potential concurrency problems that could arise.

   Is a slot free? Is a slot full?

6. Two threads contain the line of code

   ```
   x = x + 1;
   ```

   If x is a shared variable, why is this a potential problem?

   Both threads can access the variable, one reads it, then the other reads it, both update and write back (in any order), the value of x will only change by 1 even though there have been two increments.

7. Java programs are all executed by the virtual machine. Discuss whether Java threads need to be treated as if they were executing in parallel.

   Since they progress a bit at a time and the currently executing thread can be swapped at any time, you must treat them as if they were truly executed in parallel.

8. Windows 3.1 used event-driven programming to avoid the need for threads.
    a. What does “event-driven programming” mean?

The program is written as a collection of “event-handlers”, code to respond to external occurrences (stimuli or events). Whenever an event occurs the corresponding event handler is executed.

    b. You are designing a program that will allow a user to type in information that will be displayed and also sent to a remote program. The remote program can send Cntl+S and Cntl+Q to stop or permit the program to transmit. When it is permitted to transmit, the program sends the next character as soon as the local device reports that the previous character has been transmitted.
        i. Outline a design using threads

Three threads: user input, transmit, receive.

Input puts characters into buffer. Transmit checks its ok to send, takes one from the buffer, sends and waits for an indication it has been received. If it’s not ok to send Transmit waits. Receive waits for a start/stop character and sets a flag, reactivating the transmit task if necessary.

        ii. Outline a design using event-driven programming

Wait for the next event
Switch event
Key pressed
Store in buffer
If ok to send & not waiting for an ack, send the next char from the buffer
Char successfully sent
If ok to send, send the next char from the buffer
Ctrl+S
set ok to send false
Ctrl+Q
set ok to send true
if not waiting for an ack, send the next char from the buffer

        iii. Compare the two

There are complicated interactions between the tasks. Because in event-driven programming only one handler can be executed at a time, there are fewer problems with concurrency. As the complexity grows, the interaction between events is likely to become greater and so more difficult to understand than the logically separate tasks.

    c. Does event-driven programming avoid all the problems of using threads? Consider the equivalent of the producer-consumer problem described in 5.

No, what happens when the buffer is full and another “produce item” event occurs? If the event handler isn’t allowed to return until it’s dealt with the event, there will be a deadlock. If it does return, where does the item go?

9. Why might you want to use threads in the following situations?
    a. A word-processor

Print while typing, spell-check while typing, ..

    b. A spreadsheet

Calculate whilst still keeping UI active

    c. A chat program

Talk and receive simultaneously

    d. An operating system

Multiple device accesses

    e. A database program

Multiple simultaneous accesses

10. How fast can a program complete the following tasks?
    a. Calculate the square roots of N numbers using a single processor

N

    b. Calculate the square roots of N numbers using as many processors as you want

1 (each calculation has its own processor)

    c. Calculate the sum of N integers using a single processor

N

    d. Calculate the sum of N integers using two processors

N/2+1 – add half the numbers on each processor, then add the two sub totals.

    e. Calculate the sum of N integers using as many processors as you want

Sort N items using a single processor Roughly $Log_2(N)$ – Use N/2 processors to add pairs of numbers, then N/4 processors to add pairs of the results, then N/8 …until you add the last two results.

    f. Sort N items using as many processors as you want

N – use N/2 processors to compare pairs starting with number 1 and swap them if necessary, then compare pairs starting with number 2 and swap if necessary, then alternate, after N iterations, the array will be sorted.

Semaphores

11. What is a semaphore?

A non-negative integer manipulated by two atomic operations, one to increment the semaphore, one to decrement it. If the semaphore is zero, a thread making a decrement call is suspended until an increment occurs, when one of the waiting threads continues.

12. How can a semaphore be used
    a. to ensure mutual exclusion

Initialise it to one, decrement it before (acquire) and increment it after (release) the critical region.

    b. to ensure one thread doesn't proceed until another thread has completed a task

Initialise it to zero, the thread to be held back waits on the semaphore, the other releases it when it's completed the task.

13. Why is the semaphore approach to mutual exclusion more risky than the Java approach using synchronized methods?

The Java approach guarantees that you can't forget to release the lock – it is automatically freed when the synchronized method returns. With semaphores, it's possible to forget to release the semaphore (e.g. if an exception occurs in the critical region).

14. What will happen when a program with one of each of the following threads executes? Consider different initialisation.

```
class Thread1 extends Thread {
…
     run(){
       …
       sem1.acquire();
       sem2.acquire();
       …
       sem1.release();
       sem2.release();
     }
…
}
```

```
class Thread2 extends Thread {
…
     run(){
       …
       sem2.acquire();
       sem1.acquire();
       …
       sem1.release();
       sem2.release();
     }
…
}
```

If either semaphore is initialized to zero, both threads will deadlock.
If both are initialized to one, both threads could run to completion or they could deadlock (each holding one of the semaphores and waiting for the other.
Otherwise, both will run to completion

15. Evaluate the following four implementations of a semaphore using the Java Primitives

```
interface Semaphore {
    public void acquire();
    public void release();
}
```

```
class Semaphore1 implements Semaphore{
    int available;

    public Semaphore1(int initial){
        available = initial;
    }

    public void acquire(){
       while (available <= 0){
           Thread.yield();
       }
       available = available - 1;
    }

    synchronized public void release(){
      available++;
    }
}
```

```
class Semaphore2 implements Semaphore{
    int available;

    public Semaphore2(int initial){
        available = initial;
    }

    synchronized public void acquire(){
       while (available <= 0){
           Thread.yield();
       }
       available = available - 1;
    }

    synchronized public void release(){
      available++;
    }
}
```

```
class Semaphore3 implements Semaphore{
   int available;

 public Semaphore3(int initial){
    available = initial;
 }

 synchronized public void acquire(){
   while (available <= 0){
      try {
         wait();
      }
      catch (InterruptedException e){ }
   }
   available = available - 1;
 }

 synchronized public void release(){
   available++;
   notifyAll();
 }
}
```

```
class Semaphore4 implements Semaphore{
 int available;

 public Semaphore4(int initial){
     available = initial;
 }

 synchronized public void acquire(){
    if (available <= 0){
       try {
           wait();
       }
       catch (InterruptedException e){ }
    }
       available = available - 1;
 }

 synchronized public void release(){
   available++;
   notifyAll();
 }
}
```

Semaphore1 *Could end up with two threads proceeding when the semaphore has a value of one – one notices it's non-zero and breaks out of the loop, then another one does, before the first can decrement it.*

Semaphore2 *Could get deadlock because even though acquire allows other threads to proceed if the available count is zero, no other thread can execute release.*

Semaphore3 *This should work. If several threads are waiting, both will be released and will be allowed into the critical region one at a time, but because of the loop, they will check that the count is positive, so one will proceed, the other will wait again.*

Semaphore4 *This won't work if there are two waiting threads, both will be released and although only one will be executing the rest of the method at a time, because they don't recheck the count, both can proceed.*

16. A binary semaphore is a semaphore with a maximum count of 1: additional signals have no effect. Assume you only have binary semaphores available. Using pseudocode, design a class to provide normal (counting) semaphores using binary semaphores to ensure thread safety.

```
class SemaphoreImpl implements Semaphore{
   int available;
   int waitingCount;
   binarySemaphore mutex = new binarySemaphore(1);
   binarySemaphore waitSem = new binarySemaphore(1);

 public SemaphoreImpl(int initial){
    available = initial;
    waitingCount = 0;
 }

 public void acquire(){
   mutex.acquire();
   if (available <= 0){
      waitingCount++;
      mutex.release();
      waitSem.acquire();
   }
   available = available - 1;
   mutex.release();
 }

 public void release(){
   mutex.acquire();
   available++;
   if (waitingCount > 0){
      waitingCount--;
      waitSem.release();
      return;
   }
   mutex.release();
 }
}
```

Deadlock

17. Explain the following program. Find the key problem with the program, explain it and fix it.

```
package philosopher;
import java.util.*;

class Chopstick {
    private static int counter = 0;
    private int number = counter++;
    public String toString() {
        return "Chopstick " + number;
    }
}

class Philosopher extends Thread {
    private static Random rand = new Random();
    private static int counter = 0;
    private int number = counter++;
    private Chopstick leftChopstick;
    private Chopstick rightChopstick;

    public Philosopher(Chopstick left, Chopstick right) {
        leftChopstick = left;
        rightChopstick = right;
        start();
    }

    public void think() {
        System.out.println(this + " thinking ");
        pause(10);
    }

    public String toString() {
        return "Philosopher " + number;
    }

    void pause(int duration){
        try {
            sleep(rand.nextInt(duration));
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public void eat(int i) {
        synchronized(leftChopstick) {
            System.out.println(this + " has "
                    + this.leftChopstick + " Waiting for "
                    + this.rightChopstick);
            pause(10);
            synchronized(rightChopstick) {
                System.out.println(this + " eating "+i);
                pause(10);
            }
        }
    }

    public void run() {
        System.out.println(this + " starting");
        for(int i = 0; i < 100; i++) {
            think();
            eat(i);
        }
        System.out.println(this + " finished");
    }
}
```

```
public class DiningPhilosophers {
    public static void main(String[] args) {
        Philosopher[] philosopher = new Philosopher[5];
        Chopstick left = new Chopstick();
        Chopstick right = new Chopstick();
        Chopstick first = left;

        for(int i =0; i < philosopher.length; i++){
            philosopher[i] = new Philosopher(left, right);
            left = right;
            if (i < philosopher.length-1)
                right = new Chopstick();
        }
        philosopher[philosopher.length - 1] = new Philosopher(right, first);

        for(int i =0; i < philosopher.length; i++){
            try {
                philosopher[i].join();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("all done");
    }
}
```

Note: System.out.println(this) is equivalent to System.out.println(this.toString())

The key problem is

```
philosopher[philosopher.length - 1] = new Philosopher(right, first);
```

this ends up with the philosophers possibly being in a loop because of the way that chopsticks are allocated and the way that the left is always taken first. The fix is simple:

```
philosopher[philosopher.length - 1] = new Philosopher(first, right);
```

This is equivalent to ensuring that the chopsticks are numbered and the lower number is always taken before the higher.

18. a) There are four conditions required for deadlock to occur. What are they?

Resources can't be taken away from threads by force
Resources can be claimed a few at a time without releasing existing ones
A cycle of threads occurs with each waiting for threads to be released by the next
Resources can't be shared

b) Which condition did you prevent when you fixed the philosophers' problem?

Cycle of threads can't be set up

19. How could you modify the program to allow the main program to stop the philosophers thinking after say 30 seconds (you will need to remove the limit of 100 iterations of thinking and eating).

Create a field, requestStop, in each philosopher, set to false. This should be checked each time around the run loop. If it is true, halt the philosopher. Provide a way for the main program to set this field. If you want to allow threads to terminate early, you would also have to use another thread to wait for 30 seconds before killing the philosophers. This task would have to be interrupted by the main program after all the philosophers have stopped.

20. Is the solution fair? Could a philosopher starve?

The solution is fair. The simple answer is that no philosopher because the philosophers are all finite and eventually everyone will have a chance to run. If you take a looser view of starvation that includes a philosopher being stopped from running until the others have finished,it's more interesting .
Because eating and thinking take some time, it would not be possible for the other philosophers to keep the system so busy that one philosopher doesn't have the chance to request a chopstick. Once it's got one, it will get the other.

If there were no fixed time delays, it would be possible for a thread to starve – the others could run to completion before it starts. However if the threads get time one after another, starvation isn't possible.

21. What does "starvation" mean when discussing a concurrent solution?

Being able to run without getting processor time.

Reference: https://archive.org/details/TIJ4CcR1

Higher level concurrency primitives

22. The **collection class**, Hashtable, was introduced in JDK 1.0 to provide a **thread-safe**, **associative map**. To ensure thread-safety, all methods of Hashtable were **synchronized**. HashMap, introduced in the Collections framework of JDK 1.2, addressed thread-safety by providing an unsynchronized base class and a synchronized **wrapper**, Collections.synchronizedMap.
    a. Explain the meaning of the bold terms.
    b. Give two reasons why programmers might regard Hashtable as inefficient.
    c. Why is HashMap an improvement?

23. The synchronized collection wrappers, synchronizedMap and synchronizedList, are conditionally thread-safe -- all individual operations are thread-safe, but sequences of operations may be subject to data races.
    a. What problem can occur with the following code?

```
Map m = Collections.synchronizedMap(new HashMap());

// code to obtain a key and the corresponding value

if (!m.containsKey(key))
m.put(key, value);
```

    b. What problem can occur with the following code to process each item in a list?

```
List l = Collections.synchronizedList(new ArrayList());

// code to initialise the list

for (int i=0; i<l.size(); i++) {
doSomething(l.get(i));
}
```

    c. What is meant by a “data race”?

24. A proposed implementation of a map makes use of a linked list of nodes, with each node pointing to (referencing) a key, the corresponding value, and the next node in the list. Discuss whether using a lock on each node would provide greater performance under low and high load (lots of threads adding new key-value pairs, modifying values for the same key, or looking up keys, with a much higher proportion of look-up operations, and very few deletions). Is it still conditionally thread-safe?

25. If a class has two synchronised methods (e.g. printAllItems() and printAnItem()) with one method calling the other, will the system deadlock in the following conditions?
    a. when a thread calls printAllItems() – because when it internally calls printAnItem(), the lock is already held.
    b. when a thread is executing printAllItems(), another thread calls printAnItem(), and is suspended, then the first thread calls printAnItem() as part of its call to printAllItems().

26. A ReentrantLock is provided as part of java.util.concurrent. One of the arguments to the constructor of ReentrantLock is a boolean value that lets you choose whether you want a fair or an unfair lock.
    a. What is meant by a “fair lock”?
    b. Why would you want an “unfair lock”?

27. Many server applications, such as Web servers, database servers, file servers, or mail servers, are oriented around processing a large number of short tasks that arrive from some remote source. Compare the use of a thread pool with the creation of a new thread to service each request.

28. Can you see any problems with a thread pool class in which a client thread would wait for an available pool thread and pass the job to that thread for execution? Suggest an improvement.

Exceptions

Answer the following questions based on the following (slightly edited) article in **Appendix A** (https://www.ibm.com/developerworks/java/library/j-jtp05236/index.html, accessed 01/19).

29. What is an exception?
30. What does “throws an exception” mean?
31. What is a **checked exception**?
32. What is the difference between `sleep()` and `wait()`?
33. What is a **blocking method**?
34. What is a **cancellable** method?
35. How is the “thread's interruption status” recorded?
36. What does “poll” mean?
37. What is the difference between `Thread.isInterrupted()` and `Thread.interrupted()`?
38. What does “Interruption is a cooperative mechanism” mean?
39. Why may ignoring interruption requests compromise responsiveness?
40. Why might we not want an activity to stop immediately?
41. Why is a method that calls a blocking method itself a blocking method?
42. What is a “throws clause”?
43. How do you “re-throw” an exception?
44. What does “swallowing an interrupt” mean?
45. Why is throwing InterruptedException not an option when a task defined by Runnable calls an interruptible method?
46. Why re-interrupt the current thread if you can’t re-throw the interrupt?
47. Is it normal for a method to respond immediately to an interrupt?
48. Does the latest version of Java have templates?
49. What problems are caused if blocking methods don’t throw InterruptedException? Give examples of “work-arounds”
50. Summarise the article. (Don’t just copy the summary).

Java theory and practice: Dealing with InterruptedException

You caught it, now what are you going to do with it?

Brian Goetz, Principal Consultant, Quiotix

23 May 2006

Java language methods, such as Thread.sleep() and Object.wait(), throw InterruptedException. Why?

Because it is a checked exception, Java insists that the method is in a try…catch block. The most common response to InterruptedException is to catch it and do nothing. Unfortunately, this approach throws away important information about the fact that an interrupt occurred, which could compromise the application's ability to cancel activities or shut down in a timely manner.

Blocking methods

When a method throws InterruptedException, it is telling you that it is a blocking method and that it will make an attempt to unblock and return early - if you ask nicely.

A blocking method is different from an ordinary method that just takes a long time to run. The completion of an ordinary method is dependent only on how much work you've asked it to do and whether adequate computing resources (CPU cycles and memory) are available. The completion of a blocking method, on the other hand, is also dependent on some external event, such as timer expiration, I/O completion, or the action of another thread (releasing a lock, setting a flag, or placing a task on a work queue). Ordinary methods complete as soon as their work can be done, but blocking methods are less predictable because they depend on external events. Blocking methods can compromise responsiveness because it can be hard to predict when they will complete.

Because blocking methods can potentially take forever if the event they are waiting for never occurs, it is often useful for blocking operations to be cancellable. (It is often useful for long-running non-blocking methods to be cancellable as well.) A cancellable operation is one that can be externally moved to completion in advance of when it would ordinarily complete on its own. The interruption mechanism provided by Thread and supported by Thread.sleep() and Object.wait() is a cancellation mechanism; it allows one thread to request that another thread stop what it is doing early. When a method throws InterruptedException, it is telling you that if the thread executing the method is interrupted, it will make an attempt to stop what it is doing and return early and indicate its early return by throwing InterruptedException. Well-behaved blocking library methods should be responsive to interruption and throw InterruptedException so they can be used within cancellable activities without compromising responsiveness.

Thread interruption

Every thread has a Boolean property associated with it that represents its interrupted status. The interrupted status is initially false; when a thread is interrupted by some other thread through a call to Thread.interrupt(), one of two things happens. If that thread is executing a low-level interruptible blocking method like Thread.sleep(), Thread.join(), or Object.wait(), it unblocks and throws InterruptedException. Otherwise, interrupt() merely sets the thread's interruption status. Code running in the interrupted thread can later poll the interrupted status to see if it has been requested to stop what it is doing; the interrupted status can be read with Thread.isInterrupted() and can be read and cleared in a single operation with the poorly named Thread.interrupted().

Interruption is a cooperative mechanism. When one thread interrupts another, the interrupted thread does not necessarily stop what it is doing immediately. Instead, interruption is a way of politely asking another thread to stop what it is doing if it wants to, at its convenience. Some methods, like Thread.sleep(), take this request seriously, but methods are not required to pay attention to interruption. Methods that do not block but that still may take a long time to execute can respect requests for interruption by polling the interrupted status and return early if interrupted. You are free to ignore an interruption request, but doing so may compromise responsiveness.

One of the benefits of the cooperative nature of interruption is that it provides more flexibility for safely constructing cancellable activities. We rarely want an activity to stop immediately; program data structures could be left in an inconsistent state if the activity were canceled mid-update. Interruption allows a cancellable activity to clean up any work in progress, restore invariants, notify other activities of the cancellation, and then terminate.

Dealing with InterruptedException

If throwing InterruptedException means that a method is a blocking method, then calling a blocking method means that your method is a blocking method too, and you should have a strategy for dealing with InterruptedException. Often the easiest strategy is to throw InterruptedException yourself, as shown in the putTask() and getTask() methods in Listing 1. Doing so makes your method responsive to interruption as well and often requires nothing more than adding InterruptedException to your throws clause.

Listing 1. Propagating InterruptedException to callers by not catching it

```
public class TaskQueue {
    private static final int MAX_TASKS = 1000;

    private BlockingQueue<Task> queue
        = new LinkedBlockingQueue<Task>(MAX_TASKS);

    public void putTask(Task r) throws InterruptedException {
        queue.put(r);
    }

    public Task getTask() throws InterruptedException {
        return queue.take();
    }
}
```

Sometimes it is necessary to do some amount of cleanup before propagating the exception. In this case, you can catch InterruptedException, perform the cleanup, and then re-throw the exception. Listing 2, a mechanism for matching players in an online game service, illustrates this technique. The matchPlayers() method waits for two players to arrive and then starts a new game. If it is interrupted after one player has arrived but before the second player arrives, it puts that player back on the queue before re-throwing the InterruptedException, so that the player's request to play is not lost.

Listing 2. Performing task-specific cleanup before re-throwing InterruptedException

```
public class PlayerMatcher {
  private PlayerSource players;

  public PlayerMatcher(PlayerSource players) {
    this.players = players;
  }

  public void matchPlayers() throws InterruptedException {
    try {
       Player playerOne, playerTwo;
       while (true) {
         playerOne = playerTwo = null;
         // Wait for two players to arrive and start a new game
         playerOne = players.waitForPlayer(); // could throw IE
         playerTwo = players.waitForPlayer(); // could throw IE
           startNewGame(playerOne, playerTwo);
       }
    }
    catch (InterruptedException e) {
    // If interrupted with one player, put the player back
      if (playerOne != null)
        players.addFirst(playerOne);
        // Then propagate the exception
        throw e;
    }
  }
}
```

Don't swallow interrupts

Sometimes throwing InterruptedException is not an option, such as when a task defined by Runnable calls an interruptible method. In this case, you can't re-throw InterruptedException, but you also do not want to do nothing. When a blocking method detects interruption and throws InterruptedException, it clears the interrupted status. If you catch InterruptedException but cannot re-throw it, you should preserve evidence that the interruption occurred so that code higher up on the call stack can learn of the interruption and respond to it if it wants to. This task is accomplished by calling interrupt() to "reinterrupt" the current thread, as shown in Listing 3. At the very least, whenever you catch InterruptedException and don't re-throw it, reinterrupt the current thread before returning.

Listing 3. Restoring the interrupted status after catching InterruptedException

```
public class TaskRunner implements Runnable {
    private BlockingQueue<Task> queue;

    public TaskRunner(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
             while (true) {
                 Task task = queue.take(10, TimeUnit.SECONDS);
                 task.execute();
             }
         }
         catch (InterruptedException e) {
             // Restore the interrupted status
             Thread.currentThread().interrupt();
         }
    }
}
```

The worst thing you can do with InterruptedException is swallow it -- catch it and neither re-throw it nor reassert the thread's interrupted status. The standard approach to dealing with an exception you didn't plan for -- catch it and log it -- also counts as swallowing the interruption because code higher up on the call stack won't be able to find out about it. (Logging InterruptedException is also just silly because by the time a human reads the log, it is too late to do anything about it.) Listing 4 shows the all-too-common pattern of swallowing an interrupt:

Listing 4. Swallowing an interrupt -- don't do this

```
public class TaskRunner implements Runnable {
  private BlockingQueue<Task> queue;

  public TaskRunner(BlockingQueue<Task> queue) {
    this.queue = queue;
  }

  public void run() {
    try {
      while (true) {
        Task task = queue.take(10, TimeUnit.SECONDS);
        task.execute();
      }
    }
    catch (InterruptedException swallowed) {
    /* DON'T DO THIS - RESTORE THE INTERRUPTED STATUS INSTEAD */
    }
  }
}
```

If you cannot re-throw InterruptedException, whether or not you plan to act on the interrupt request, you still want to re-interrupt the current thread because a single interruption request may have multiple "recipients." The standard thread pool (ThreadPoolExecutor) worker thread implementation is responsive to interruption, so interrupting a task running in a thread pool may have the effect of both cancelling the task and notifying the execution thread that the thread pool is shutting down. If the task were to swallow the interrupt request, the worker thread might not learn that an interrupt was requested, which could delay the application or service shutdown.

Implementing cancellable tasks

Nothing in the language specification gives interruption any specific semantics, but in larger programs, it is difficult to maintain any semantics for interruption other than cancellation. Depending on the activity, a user could request cancellation through a GUI or through a network mechanism such as JMX or Web Services. It could also be requested by program logic. For example, a Web crawler might automatically shut itself down if it detects that the disk is full, or a parallel algorithm might start multiple threads to search different regions of the solution space and cancel them once one of them finds a solution.

Just because a task is cancellable does not mean it needs to respond to an interrupt request immediately. For tasks that execute code in a loop, it is common to check for interruption only once per loop iteration. Depending on how long the loop takes to execute, it could take some time before the task code notices the thread has been interrupted (either by polling the interrupted status with Thread.isInterrupted() or by calling a blocking method). If the task needs to be more responsive, it can poll the interrupted status more frequently. Blocking methods usually poll the interrupted status immediately on entry, throwing InterruptedException if it is set to improve responsiveness.

The one time it is acceptable to swallow an interrupt is when you know the thread is about to exit. This scenario only occurs when the class calling the interruptible method is part of a Thread, not a Runnable or general-purpose library code, as illustrated in Listing 5. It creates a thread that enumerates prime numbers until it is interrupted and allows the thread to exit upon interruption. The prime-seeking loop checks for interruption in two places: once by polling the isInterrupted() method in the header of the while loop and once when it calls the blocking BlockingQueue.put() method.

Listing 5. Interrupts can be swallowed if you know the thread is about to exit

```
public class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }

    public void cancel() { interrupt(); }
}
```

Noninterruptible blocking

Not all blocking methods throw InterruptedException. The input and output stream classes may block waiting for an I/O to complete, but they do not throw InterruptedException, and they do not return early if they are interrupted. However, in the case of socket I/O, if a thread closes the socket, blocking I/O operations on that socket in other threads will complete early with a SocketException. The nonblocking I/O classes in java.nio also do not support interruptible I/O, but blocking operations can similarly be canceled by closing the channel or requesting a wakeup on the Selector. Similarly, attempting to acquire an intrinsic lock (enter a synchronized block) cannot be interrupted, but ReentrantLock supports an interruptible acquisition mode.

Noncancellable tasks

Some tasks simply refuse to be interrupted, making them noncancellable. However, even noncancellable tasks should attempt to preserve the interrupted status in case code higher up on the call stack wants to act on the interruption after the noncancellable task completes. Listing 6 shows a method that waits on a blocking queue until an item is available, regardless of whether it is interrupted. To be a good citizen, it restores the interrupted status in a finally block after it is finished, so as not to deprive callers of the interruption request. (It can't restore the interrupted status earlier, as it would cause an infinite loop -- BlockingQueue.take() could poll the interrupted status immediately on entry and throws InterruptedException if it finds the interrupted status set.)

Listing 6. Noncancellable task that restores interrupted status before returning

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // fall through and retry
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

Summary

You can use the cooperative interruption mechanism provided by the Java platform to construct flexible cancellation policies. Activities can decide if they are cancellable or not, how responsive they want to be to interruption, and they can defer interruption to perform task-specific cleanup if returning immediately would compromise application integrity. Even if you want to completely ignore interruption in your code, make sure to restore the interrupted status if you catch InterruptedException and do not re-throw it so that the code that calls it is not deprived of the knowledge that an interrupt occurred.