1. Inherit a class from Thread and override the run() method. Inside run(), print a message, and then call sleep(). Repeat this three times, then return from run(). Put a start-up message In the constructor and override finalize() to print a shut-down message. Make a separate thread class that calls System.gc() and System.runFinalization() inside run(), printing a message as it does so. Make several thread objects of both types and run them to see what happens.

2. Write a short program in which two threads both increment a shared integer repeatedly, without proper synchronization, 1,000,000 times, printing the result at the end of the program. Modify the program to use synchronized to ensure that increments on the shared variable are atomic.

3. Write a short program in which one thread increments an integer 1,000,000 times, and a second thread prints the integer - without waiting for it to finish. Now modify the program to use a condition variable to signal completion of the addition task by the first thread before the second thread prints the value.

4. Demonstrate that a synchronized method in a class can call a second synchronized method in the same class, which can then call a third synchronized method in the same class. Create a separate Thread object that invokes the first synchronized method.

5. Create two Thread subclasses, one with a run() that starts up and then calls wait(). The other class's run() should capture the reference of the first Thread object. Its run() should call notifyAll() for the first thread after some number of seconds have passed so that first thread can print a message.

6. Create an example of a "busy wait." One thread sleeps for a while and then sets a flag to true. The second thread watches that flag inside a while loop (this is the "busy wait") and when the flag becomes true, sets it back to false and reports the change to the console. Note how much wasted time the program spends inside the "busy wait" and create a second version of the program that uses wait() instead of the "busy wait."

7. \*Semaphores are a widely used synchronization primitive -- but not one of the fundamental primitives provided by Java. Implement a counting semaphore using synchronized(), wait(), and notify() in Java.

8. \*\* Deadlocks are an inherent problem in concurrent systems using locks or other blocking primitives. Implement a deadlock involving two threads and two locks in Java.

9. \*\*\* Data races occur when there is insufficient synchronization around composite operations. Write a short program that illustrates a data race.

1. Наследуйте класс от Thread и переопределите метод run (). Внутри run () напечатайте сообщение, а затем вызовите sleep (). Повторите это три раза, затем вернитесь из run (). Поместите сообщение о запуске В конструкторе и переопределите finalize (), чтобы распечатать сообщение о завершении работы. Создайте отдельный класс потока, который вызовет System.gc () и System.runFinalization () внутри run (), печатая сообщение так, как оно это делает. Сделайте несколько объектов потока обоих типов и запустите их, чтобы увидеть, что происходит.

2. Напишите короткую программу, в которой два потока оба раза увеличивают общее целое число без надлежащей синхронизации, 1,000,000 раз, распечатывая результат в конце программы. Измените программу для использования synchronized, чтобы убедиться, что приращения в общей переменной являются атомарными.

3. Напишите короткую программу, в которой один поток будет увеличиваться на 1.000.000 раз, а второй поток выведет целое число - не дожидаясь его завершения. Теперь измените программу, чтобы использовать переменную условия, чтобы сигнализировать о завершении задачи сложения первым потоком, прежде чем второй поток напечатает значение.

4. Демонстрируйте, что синхронизированный метод в классе может вызывать второй синхронизированный метод в том же классе, который затем может вызывать третий синхронизированный метод в том же классе. Создайте отдельный объект Thread, который вызывает первый синхронизированный метод.

5. Создайте два подкласса Thread, один с run (), который запускается, а затем вызывает wait (). Класс run () другого класса должен захватить ссылку на первый объект Thread. Его run () должен вызывать notifyAll () для первого потока по истечении некоторого количества секунд, чтобы первый поток мог напечатать сообщение.

6. Создайте пример «ожидание занятости». Одна нить спит некоторое время, а затем устанавливает флаг в true. Второй поток следит за этим флагом внутри цикла while (это «ожидание занятости»), и когда флаг становится истинным, возвращает его false и сообщает об изменении на консоль. Обратите внимание на то, сколько времени тратится на выполнение программы внутри «ожидание занятости» и создайте вторую версию программы, которая использует wait () вместо «ожидание занятости».

7. \* Семафоры являются широко используемым примитивом синхронизации, но не одним из основных примитивов, предоставляемых Java. Реализация счетного семафора с использованием synchronized (), wait () и notify () в Java.

8. \*\* Тупики являются неотъемлемой проблемой в параллельных системах, использующих блокировки или другие блокирующие примитивы. Реализовать тупик, включающий два потока и два замка в Java.

9. \*\*\* Сбои данных происходят, когда недостаточная синхронизация вокруг составных операций. Напишите короткую программу, которая иллюстрирует гонку данных.