5. יש שתי פונקציות של Event שיכולים לסייע לנו במקרה זה:

הראשונה-מחלקה יכולה להרשם לאיוונט אך רק המחלקה בה מוגדר האיוונט יכולה להפעיל אותו- הבנק הוא היחיד שיודע מה ממצב הכסף בחשבון הבנק ולכן הוא היחיד שצריך לפעיל את האיוונט במצב של מינוס. לא נרצה שהכספומט או מחלקה אחרת יכלו להפעיל את האיוונט.

השנייה- אי אפשר להשתמש ב= רק ב+= או -=- זה עוזר לנו כי יכול להיות לאיוונט של המינוס רשומות מספר פונקציות (שיכולולת להיות ממחלקות שונות)- למשל הצגת שגיאה בכספומט, שליחת ממכתב הבייתה, חסימת אשראי וכו' נרצה שתיהיה אופציה להוסיף את כל הפונקציות לאיוונט אך לא נרצה שפונקצייה אחת תחליף את האחרות.

אתגר מימוש תוכנית קצרה נמצא בפרוייקט bankQuestion5

6. א. void

ב.יוצרים מחלקה חדש שיורשת מ EventArgs (משתמשים ב`EventHandler<T>`) למשל

```
public class BalanceEventArgs:EventArgs
    {
        public int Balance { get; set; }
    }
}
```

```
(   public   event EventHandler<BalanceEventArgs>  minusBalnce;)
```

8. במקרה השנחי יהיה כדאי להשמש בריבוי משימות כי יש מספר משתמשים בו זמנית אם נרצה להציג את ההודעות מהמשתמשים בו זמנית נצטרך thread ב א. אלא אם כן החישוב ממש ארוך או מסובך לא כדאי להתמש בריבוי משימות. לוקח זמ ן לייצר thread וזה לא משתלם להשתמש בו בשביל חישוב פשוט.

9.א. without threads it took 00:00:14.6371511

ב. with threads it took 00:00:00.1372137

זה היה מהיר יותר מסעיף 1 בצורה משמעותית כי בסעיף אחד התוכנית הייתה צריכה לעבור על כל לולאה לחכות שהיא תסתיים ורק אחר כך לעבור על הלולאה הבאה. בסעיף ב הלולאות קרו במקביל (והלולאות מספיק ארוכות כך שלמרות שלוקח זמן ליצור את ה threads זה עדיין משתלם)

10.Running - ה thread רץ- הוא קיבל הקצאה במעבד ומבצע את הפקודות.

Suspended-- ה thread נעצר (אך עדיין קיים) ומפנה את המקום שלו במעבד ומאפשר ל thread אחרים להתחיל ולרוץ.

Aborted - ה thread מושמד.

11.הייתי מגדירה אותו Background Thread בגלל שהייתי רוצה שה thread יוריד מידע מהאינטרנט רק כל עוד ה thread הראשי פועל. וBackground Thread מפסיק לפעול ברגע Foreground Thread מפסיק לפעול.

13. Threads מה thread pool נועדו למשימות קצרות לכן:

א. thread רגיל.

ב. thread מה thread pool

15. א. כדאי להשתמש בlock כאשר יש מצב שבו יש סכנה ששני Threads יבצעו את המשימה בו זמנית. Lock נועל את הקטע הזה וגורם לכך שרק thread אחד יכול לבצע אותו בכל פעם

ב. critical section הוא קטע ש שמסוכן שיותר מ thread אחד יבצע אותו בו זמנית.

ג. System.Threading.Monitor::Enter ו- System.Threading.Monitor::Exit

ד. כן כי lock גורם לכך שרק thread יוכל להכנס לבצע את הקט אותו הוא נועל.

16. ConcurrentQueue<T> היתרון: הוא thread safe החיסרון: הוא מורכב ולוקח לו קצת יותר זמן לרוץ.

17. dead lock

18. אשתמש ב Wait / Pulse וב (Monitor.Enter() + Monitor.Exit(). אפשר לממש את זה כמו שממשים בשאלה הבאה יוצרים שתי פונקציות אחת – אחת לרופא ואחת לאחות. ברופא נשים wait ואחרי ה wait נקרא לאחות ובאחות pulse בשניהם הקוד יהיה נעול ב + Monitor.Enter() Monitor.Exit() עם אותו מפתח.מה שיקרה שה thread יכנס אל הרופא אם המפתח פנוי הוא יחכה לטיפול ויחזיר את המפתח למקום ל thread הבא. נעשה pulse ל thread הראשון שמחכה המטופל יחזור מההמתנה לקבל טיפול אצל הרופא ויעבור לאחות. כשיסיים את הטיפול אצל האחות יקרא למטופל הבא ואם יש מטופל שמחכה עוד מטופל ישחורר.

21. א. Auto
ב. Manual

23. א. - Task יכולים להחזיר תוצאה thread לא
- Task פשוטים לשימוש
- אפשר לשרשר מספר Task ביחד

ב. NET Framework 4.

24. א. נכון

ב. נכון.

25.א. יש להשתמש באופציה TaskCreationOptions.LongRunning בצורה הבאה:

```
Task task = Task.Factory.StartNew (() => ...,
TaskCreationOptions.LongRunning);
```

ב.

```
Task task = Task.Factory.StartNew(() => { Thread.Sleep(10000); },
                                        TaskCreationOptions.LongRunning);
```

.26

```
int result =Task<int>.Factory.StartNew((object o) =>
          {
              int[] numbers = o as int[];
              return numbers[0] + numbers[1];

                                                  }, new int[] { 2, 3 }).Result;
```

27.א. ניתן להשתמש ב continueWith

ב.

```
                                Task.Run(() => { Console.WriteLine("Welcome");
                  }).ContinueWith((antecedent) =>Console.WriteLine("Goodbye"));
```

ג. נשתמש ב TaskContinuationOptions.OnlyOnFaulted:

```
Task.Run(() => { Console.WriteLine("Welcome"); throw new Exception(); }).
                                                        ContinueWith((antecedent)
          =>Console.WriteLine("Goodbye"),TaskContinuationOptions.OnlyOnFaulted);
```

.28

```
Task.Run(() => { Console.WriteLine($"the is: {DateTime.Now}");}).
               ContinueWith((antecedent) => Console.WriteLine($"the is:
{DateTime.Now}"), TaskContinuationOptions.OnlyOnRanToCompletion);;
```

29. א. נשתמש Task.waitall(thread1,thread2.....) כשה threads שאנחנו שולחים לפונקציה הם אלו שאנחנו רוצים לחכות להם.

ב.

```
 Task t1 = new Task(() => Thread.Sleep(5000));
          Task t2 = new Task(() => Thread.Sleep(5000));
          Task t3 = new Task(() => Thread.Sleep(5000));

          t1.Start();
          t2.Start();
          t3.Start();


          while (!t1.IsCompleted && !t2.IsCompleted && !t3.IsCompleted)
```

```
            {
                Thread.Sleep(1000);
            }

            Console.WriteLine("All tasks are done");
```

ג. כי זה גורם לכך שהתוכנית תבדוק בתדירות גבוהה מאוד אם ה Task הסתיים וזה מזבזבז הרבה משאבים. אפשר לפתור את הבעיה הזאת אם נשתמש ב- Thread.Sleep בתוך לולאת ה while

30.א. מחכה ש task כלשהו מאלו ששלחנו לה יסיים

ב.

```
            Random random = new Random();
            Task[] tasks = new Task[3];
            for (int i = 0; i < 3; i++)
            {
                tasks[i] = Task.Run(()=>Thread.Sleep(random.Next(5000,
10000)));
            }

            Task.WaitAny(tasks);

                                Console.WriteLine("one task is done!");
```

31. א.

```
int num = 0;
Task<int>.Run(() => {  return 6 / num; });
```

התוכנית לא קרסה

ב.

```
  int num = 0;
  int result= Task<int>.Run(() => {  return 6 / num; }).Result;
```

התוכנית קרסה

ג.

```
  int num = 0;
            try
            {
```

```
        int result = Task<int>.Run(() => { return 6 / num; }).Result;
    }
    catch (AggregateException e)
    {
        Console.WriteLine("cannot divide by zero!");

    }
```

ד. AggregateException

32. א. ההבדל בניהם הוא שכשאר אנו מבצעים את suspend ל thread ה thread מושהה לא משנה באמצע איזה פעולה הוא נמצא ובלי שנצטרך לבדוק מתוך ה thread אם בוצעה השהיה. לעומת זאת כשאנו מבצעים ביטול ל task אנו צריכים לשלוח cancellation token, לבקש את ביטול ה task ולבדוק מתוך ה task האם הייתה בקשה לביטול .מנגנון הביטול של ה task עדיף כי אנחנו יכולים לשלוט על הביטול- לקבוע מתי הוא יקרה ואיך להגיב לו. (אם נשתמש ב suspend של thread- אנחנו יכולים להגיע למשל למצב של dead lock כאשר ה threadנמצא באמצע critical sectionשנעול במפתח- אם יושהה בזמן שהוא מחזיק במפתח, הוא ימשיך להחזיק במפתח ויכול לתקוע threads אחרים שזקוקים למפתח)

ב. cancelation token:

```
CancellationTokenSource source = new CancellationTokenSource();
CancellationToken token = source.Token;
```

35. async יוצרת מכונת מצבים היא רצה בצורה סנכרונית עד שהיא פוגשת await אז היא מושהית עד שהawaited task מושלם בנתיים השליטה חוזרת לפנוקציה שממנה נקראה.

36.א. async

ב. לא

37.נשתמש ב async await במקרים הבאים:

I/O-bound work- הקוד קורא קובץ, מחכה לשרת אינטרנט וכדומה

CPU-bound work- הקוד מבצע חישוב מורכב

39. כי בכל פעם שאנחנו משתמשים במילה async נוצרת מכונת מצבים ברקע ומורכבות הזאת לא תמיד משתלמת.