```
 1: package gotocon;
 2:
 3: import java.util.*;
 4:
 5: public class Game {
 6:
 7:     private Player winner;
 8:
 9:     public enum Score {
10:         ZERO {
11:             @Override
12:             public Score next() {
13:                 return Score.FIFTEEN;
14:             }
15:         },
16:         FIFTEEN {
17:             @Override
18:             public Score next() {
19:                 return Score.THIRTY;
20:             }
21:         },
22:         THIRTY {
23:             @Override
24:             public Score next() {
25:                 return Score.FORTY;
26:             }
27:         },
28:         FORTY {
29:             @Override
30:             public Score next() {
31:                 return Score.ADVANTAGE;
32:             }
33:         },
34:         ADVANTAGE {
35:             @Override
36:             public Score next() {
37:                 throw new UnsupportedOperationException();
38:             }
39:         };
40:
41:         public abstract Score next();
42:     }
43:
44:     private Map<Player, Score> scores = new HashMap<Player, Score>();
45:
46:     public Game(Player player1, Player player2) {
47:         scores.put(player1, Score.ZERO);
48:         scores.put(player2, Score.ZERO);
49:     }
50:
51:     public Player getWinner() {
52:         return winner;
53:     }
54:
55:     public Score getScore(Player player) {
56:         return scores.get(player);
57:     }
58:
59:     public void scorePointForPlayer(Player player) {
60:         if (isGameFinished()) {
61:             throw new IllegalStateException("Cannot continue finished game");
62:         }
63:         Score currentScore = getScore(player);
64:         if (currentScore == Game.Score.FORTY) {
65:             if (isGameAtDeuce()) {
```

```
66:                 scores.put(player, currentScore.next());
67:             } else if (gameHasAdvantage()) {
68:                 reduceToDeuce();
69:             } else {
70:                 winner = player;
71:             }
72:         } else if (currentScore == Game.Score.ADVANTAGE) {
73:             winner = player;
74:         } else {
75:             scores.put(player, currentScore.next());
76:         }
77:     }
78:
79:     private boolean gameHasAdvantage() {
80:         for (Score s : scores.values()) {
81:             if (s == Score.ADVANTAGE) {
82:                 return true;
83:             }
84:         }
85:         return false;
86:     }
87:
88:     private void reduceToDeuce() {
89:         for (Player p : scores.keySet()) {
90:             scores.put(p, Score.FORTY);
91:         }
92:     }
93:
94:     private boolean isGameAtDeuce() {
95:         for (Score s : scores.values())
96:         {
97:             if (s != Score.FORTY) {
98:                 return false;
99:             }
100:        }
101:        return true;
102:    }
103:
104:    private boolean isGameFinished() {
105:        return winner != null;
106:    }
107: }
```

```
1: package gotocon;
2:
3: public class Player {
4:
5: }
```

```java
package gotocon;

import gotocon.Game.Score;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class TennisRulesTest {

    private Player player1;
    private Player player2;
    private Game game;

    @Before
    public void before() {
        player1 = new Player();
        player2 = new Player();
        game = new Game(player1, player2);
    }

    @Test
    public void happyPath() {
        scorePointForPlayer(game, player1);
        assertScore(Score.FIFTEEN, Score.ZERO);

        scorePointForPlayer(game, player1);
        assertScore(Score.THIRTY, Score.ZERO);

        scorePointForPlayer(game, player1);
        assertScore(Score.FORTY, Score.ZERO);

        scorePointForPlayer(game, player1);
        Assert.assertEquals(player1, game.getWinner());
    }

    @Test
    public void complainsIfContinuingFinishedGame() {
        scorePointForPlayer(game, player1);
        scorePointForPlayer(game, player1);
        scorePointForPlayer(game, player1);
        scorePointForPlayer(game, player1);

        try {
            scorePointForPlayer(game, player1);
            Assert.fail("player 1 can't win");
        } catch (IllegalStateException e) {
            // expected
        }
        try {
            scorePointForPlayer(game, player2);
            Assert.fail("player 2 can't win");
        } catch (IllegalStateException e) {
            // expected
        }
    }

    @Test
    public void handlesWinningAfterTightScores() {
        withTiedGame();

        scorePointForPlayer(game, player1);
        assertScore(Score.ADVANTAGE, Score.FORTY);
        scorePointForPlayer(game, player1);
        Assert.assertEquals(player1, game.getWinner());
    }
```

```
66:
67:     @Test
68:     public void handlesGoingBackToDeuceAfterAdvantage() {
69:         withTiedGame();
70:         scorePointForPlayer(game, player1);
71:         scorePointForPlayer(game, player2);
72:         assertScore(Score.FORTY, Score.FORTY);
73:     }
74:
75:     private void withTiedGame() {
76:         scorePointForPlayer(game, player1);
77:         scorePointForPlayer(game, player1);
78:         scorePointForPlayer(game, player1);
79:
80:         scorePointForPlayer(game, player2);
81:         scorePointForPlayer(game, player2);
82:         scorePointForPlayer(game, player2);
83:     }
84:
85:
86:
87:     private void assertScore(Score p1score, Score p2score) {
88:         Assert.assertEquals(p1score, game.getScore(player1));
89:         Assert.assertEquals(p2score, game.getScore(player2));
90:     }
91:
92:     private void scorePointForPlayer(Game game, Player winner) {
93:         game.scorePointForPlayer(winner);
94:     }
95: }
```