```java
// TestThreads.java: Define threads using the Thread class
public class TestThreads
{
  // Main method
  public static void main(String[] args)
  {
    // Create threads
    PrintChar printA = new PrintChar('a', 100);
    PrintChar printB = new PrintChar('b', 100);
    PrintNum  print100 = new PrintNum(100);

    // Start threads
    print100.start();
    printA.start();
    printB.start();

    // Pause
    System.out.println("Press Ctrl+C to close this window ...");
    MyInput.readInt();
  }
}

// The thread class for printing a specified character in specified times
class PrintChar extends Thread
{
  private char charToPrint;          // The character to print
  private int times;                 // The times to repeat

  // Construct a thread with specified character and number of
  // times to print the character
  public PrintChar(char c, int t)
  {
    charToPrint = c;
    times = t;
  }

  // Vverride the run() method to tell the system what the thread will do
  public void run()
  {
    for (int i=1; i < times; i++)
      System.out.print(charToPrint);
  }
}
```

```
// The thread class for printing number from 1 to n for a given n
class PrintNum extends Thread
{
  private int lastNum;

  // Construct a thread for print 1, 2, ... i
  public PrintNum(int n)
  {
    lastNum = n;
  }

  public void run()
  {
    for (int i=1; i <= lastNum; i++)
      System.out.print(" " + i);
  }
}

// CurrentTimeApplet.java: Display a still clock on the applet
import java.awt.*;
import java.util.*;
import javax.swing.*;

public class CurrentTimeApplet extends JApplet
{
  protected Locale locale;
  protected TimeZone tz;
  protected StillClock stillClock;
  private boolean isStandalone = false;

  // Construct the applet
  public CurrentTimeApplet()
  { }

  // Initialize the applet
  public void init()
  {
    // Load native fonts. Uncomment the following two statements,
    // if native fonts such as Chinese fonts are not used
    // GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
    // ge.getAllFonts();

    if (!isStandalone)
    {
      // Get locale and timezone from HTML
      getHTMLParameters();
    }
    // Add the clock to the applet
    createClock();
  }
```

```
// Create a clock and add it to the applet
public void createClock()
{    getContentPane().add(stillClock = new StillClock(locale, tz));
}

public void getHTMLParameters()
{
  // Get parameters from the HTML
  String language = getParameter("language");
  String country = getParameter("country");
  String timezone = getParameter("timezone");
  // Set default values if parameters are not given in the HTML file
  if (language == null)       language = "en";
  if (country == null)        country = "US";
  if (timezone == null)       timezone = "CST";
  // Set locale and timezone
  locale = new Locale(language, country);
  tz = TimeZone.getTimeZone(timezone);
}

// Main method with three auguments:
// args[0]: language such as en
// args[1]: country such as US
// args[2]: timezone such as CST
public static void main(String[] args)
{
  // Create a frame
  MyFrameWithExitHandling frame = new MyFrameWithExitHandling(
    "Display Current Time");

  // Create an instance of the applet
  CurrentTimeApplet applet = new CurrentTimeApplet();

  // It runs as an application
  applet.isStandalone = true;

  // Get parameters from the command line
  applet.getCommandLineParameters(args);

  // Add the applet instance to the frame
  frame.getContentPane().add(applet, BorderLayout.CENTER);

  // Invoke init() and start()
  applet.init();
  applet.start();

  // Display the frame
  frame.setSize(300, 300);
  frame.setVisible(true);
}
```

```
  // Get command line parameters
  public void getCommandLineParameters(String[] args)
  {
    // Declare locale and timezone with default values
    locale = Locale.getDefault();
    tz = TimeZone.getDefault();

    // Check usage and get language, country and time zone
    if (args.length > 3)
    {
      System.out.println(
        "Usage: java CurrentTimeApplet language country timezone");
      System.exit(0);
    }
    else if (args.length == 3)
    {
      locale = new Locale(args[0], args[1]);
      tz = TimeZone.getTimeZone(args[2]);
    }
    else if (args.length == 2)
    {
      locale = new Locale(args[0], args[1]);
      tz = TimeZone.getDefault();
    }
    else if (args.length == 1)
    {
      System.out.println(
        "Usage: java DisplayTime language country timezone");
      System.exit(0);
    }
    else
    {
      locale = Locale.getDefault();
      tz = TimeZone.getDefault();
    }
  }
}
```

```
// StillClock.java: Display a clock in JPanel
import java.awt.*;
import java.util.*;
import java.text.*;
import javax.swing.*;

public class StillClock extends JPanel
{
  protected TimeZone tz = TimeZone.getDefault();
  protected int xCenter, yCenter;
  protected int clockRadius;
  protected DateFormat myFormat;

  public StillClock()
  {
  }

  public StillClock(Locale locale, TimeZone tz)
  {
    setLocale(locale);
    this.tz = tz;
  }

  // Set timezone using a time zone id such as "CST"
  public void setTimeZoneID(String newTimeZoneID)
  {
    tz = TimeZone.getTimeZone(newTimeZoneID);
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);

    // Initialize clock parameters
    clockRadius =
      (int)(Math.min(getSize().width, getSize().height)*0.7*0.5);
    xCenter = (getSize().width)/2;
    yCenter = (getSize().height)/2;

    // Draw circle
    g.setColor(Color.black);
    g.drawOval(xCenter - clockRadius,yCenter - clockRadius,
      2*clockRadius, 2*clockRadius);
    g.drawString("12",xCenter-5, yCenter-clockRadius);
    g.drawString("9",xCenter-clockRadius-10,yCenter+3);
    g.drawString("3",xCenter+clockRadius,yCenter+3);
    g.drawString("6",xCenter-3,yCenter+clockRadius+10);

    // Get current time using GregorianCalendar
    GregorianCalendar cal = new GregorianCalendar(tz);
```

```
    // Draw second hand
    int second = (int)cal.get(GregorianCalendar.SECOND);
    int sLength = (int)(clockRadius*0.9);
    int xSecond =
      (int)(xCenter + sLength*Math.sin(second*(2*Math.PI/60)));
    int ySecond =
      (int)(yCenter - sLength*Math.cos(second*(2*Math.PI/60)));
    g.setColor(Color.red);
    g.drawLine(xCenter, yCenter, xSecond, ySecond);

    // Draw minute hand
    int minute = (int)cal.get(GregorianCalendar.MINUTE);
    int mLength = (int)(clockRadius*0.75);
    int xMinute =
      (int)(xCenter + mLength*Math.sin(minute*(2*Math.PI/60)));
    int yMinute =
      (int)(yCenter - mLength*Math.cos(minute*(2*Math.PI/60)));
    g.setColor(Color.blue);
    g.drawLine(xCenter, yCenter, xMinute, yMinute);

    // Draw hour hand
    int hour = (int)cal.get(GregorianCalendar.HOUR_OF_DAY);
    int hLength = (int)(clockRadius*0.6);
    int xHour = (int)(xCenter +
      hLength*Math.sin((hour+minute/60.0)*(2*Math.PI/12)));
    int yHour = (int)(yCenter -
      hLength*Math.cos((hour+minute/60.0)*(2*Math.PI/12)));
    g.setColor(Color.green);
    g.drawLine(xCenter, yCenter, xHour, yHour);

    // Set display format in specified style, locale and timezone
    myFormat = DateFormat.getDateTimeInstance
      (DateFormat.MEDIUM, DateFormat.LONG, getLocale());
    myFormat.setTimeZone(tz);

    // Display current date
    g.setColor(Color.red);
    String today = myFormat.format(cal.getTime());
    FontMetrics fm = g.getFontMetrics();
    g.drawString(today, (getSize().width -
    fm.stringWidth(today))/2, yCenter+clockRadius+30);
  }
}
```

```
// Clock.java: Show a running clock on the panel
import java.util.*;
public class Clock extends StillClock implements Runnable
{
  // Declare a thread for running the clock
  private Thread timer = null;

  // Determine if the thread is suspended
  private boolean suspended = false;

  // Default constructor
  public Clock()
  {
    super();
    // Create the thread
    timer = new Thread(this);
    // Start the thread
    timer.start();
  }

  // Construct a clock with specified locale and time zone
  public Clock(Locale locale, TimeZone tz)
  {
    super(locale, tz);

    // Create the thread
    timer = new Thread(this);

    // Sstart the thread
    timer.start();
  }

  // Implement the run() method to dictate what the thread will do
  public void run()
  {
    while (true)
    {
      repaint();
      try
      {
        timer.sleep(1000);
        synchronized (this)
        {
          while (suspended)
            wait();
        }
      }
      catch (InterruptedException ex) {  }
    }
  }
```

```
  // Resume the clock
  public synchronized void resume()
  {
    if (suspended)
    {
      suspended = false;
      notify();
    }
  }

  // Suspend the clock
  public synchronized void suspend()
  {
    suspended = true;
  }
}
```

```
// ClockGroup.java: Display a group of international clocks
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class ClockGroup extends JApplet implements ActionListener
{
  // Declare three clock panels
  private ClockPanel clockPanel1, clockPanel2, clockPanel3;

  // Declare group control buttons
  private JButton jbtResumeAll, jbtSuspendAll;

  // This main method enables the applet to run as an application
  public static void main(String[] args)
  {
    // Create a frame
    MyFrameWithExitHandling frame = new MyFrameWithExitHandling(
      "Clock Group Demo");

    // Create an instance of the applet
    ClockGroup applet = new ClockGroup();

    // Add the applet instance to the frame
    frame.getContentPane().add(applet, BorderLayout.CENTER);

    // Invoke init() and start()
    applet.init();
    applet.start();

    // Display the frame
    frame.setSize(600, 300);
    frame.setVisible(true);
  }

  // Initialize the applet
  public void init()
  {
    // Panel p1 for holding three clocks
    JPanel p1 = new JPanel();
    p1.setLayout(new GridLayout(1, 3));

    // Create a clock for Berlin
    p1.add(clockPanel1 = new ClockPanel());
    clockPanel1.setTitle("Berlin");
    clockPanel1.clock.setTimeZoneID("ECT");
    clockPanel1.clock.setLocale(Locale.GERMAN);
```

```
    // Create a clock for San Francisco
    p1.add(clockPanel2 = new ClockPanel());
    clockPanel2.clock.setLocale(Locale.US);
    clockPanel2.clock.setTimeZoneID("PST");
    clockPanel2.setTitle("San Francisco");

    // Create a clock for Taipei
    p1.add(clockPanel3 = new ClockPanel());
    clockPanel3.setTitle("Paris");
    clockPanel3.clock.setLocale(Locale.FRANCE);

    // Panel p2 for holding two group control buttons
    JPanel p2 = new JPanel();
    p2.setLayout(new FlowLayout());
    p2.add(jbtResumeAll = new JButton("Resume All"));
    p2.add(jbtSuspendAll = new JButton("Suspend All"));

    // Add panel p1 and p2 into the applet
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(p1, BorderLayout.CENTER);
    getContentPane().add(p2, BorderLayout.SOUTH);

    // Register listeners
    jbtResumeAll.addActionListener(this);
    jbtSuspendAll.addActionListener(this);
  }

  // Handlers for group control buttons
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == jbtResumeAll)
    {
      // Start all clocks
      clockPanel1.resume();
      clockPanel2.resume();
      clockPanel3.resume();
    }
    else if (e.getSource() == jbtSuspendAll)
    {
      // Stop all clocks
      clockPanel1.suspend();
      clockPanel2.suspend();
      clockPanel3.suspend();
    }
  }
}
```

```
// ClockPanel for holding a header, a clock, and control buttons
class ClockPanel extends JPanel implements ActionListener
{
  // Header title of the clock panel
  private JLabel jlblTitle;

  protected Clock clock  = null;

  // Individual clock Resume and Suspend control buttons
  private JButton jbtResume, jbtSuspend;

  // Constructor
  public ClockPanel()
  {
    // Panel jpButtons for grouping buttons
    JPanel jpButtons = new JPanel();
    jpButtons.add(jbtResume = new JButton("Resume"));
    jpButtons.add(jbtSuspend = new JButton("Suspend"));

    // Set BorderLayout for the ClockPanel
    setLayout(new BorderLayout());

    // Add title label to the north of the panel
    add(jlblTitle = new JLabel(), BorderLayout.NORTH);
    jlblTitle.setHorizontalAlignment(JLabel.CENTER);

    // Add the clock to the center of the panel
    add(clock = new Clock(), BorderLayout.CENTER);

    // Add jpButtons to the south of the panel
    add(jpButtons, BorderLayout.SOUTH);

    // Register ClockPanel as a listener to the buttons
    jbtResume.addActionListener(this);
    jbtSuspend.addActionListener(this);
  }

  // Set label on the title
  public void setTitle(String title)
  {
    jlblTitle.setText(title);
  }
```

```
  // Haandlers for buttons "Resume" and "Suspend"
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == jbtResume)
    {
      clock.resume();
    }
    else if (e.getSource() == jbtSuspend)
    {
      clock.suspend();
    }
  }

  // Resume the clock
  public void resume()
  {
    if (clock != null) clock.resume();
  }

  // Suspend the clock
  public void suspend()
  {
    if (clock != null) clock.suspend();
  }
}
```

```
// ClockApplet.java: Display a running clock on the applet
import java.applet.*;
import java.awt.*;
import java.util.*;

public class ClockApplet extends CurrentTimeApplet
  implements Runnable
{
  // Declare a thread for running the clock
  private Thread thread = null;

  // Determine if the thread is suspended
  private boolean suspended = false;

  // Initialize applet
  public void init()
  {
    super.init();

    // Create the thread
    thread = new Thread(this);

    // Start the thread
    thread.start();
  }

  // Implement the start() method to resume the thread
  public void start()
  {
    resume();
  }

  // Implement the run() method to dictate what the thread will do
  public void run()
  {
    while (true)
    {
      // Repaint the clock to display current time
      stillClock.repaint();

      try
      {
        thread.sleep(1000);
        synchronized (this)
        {
          while (suspended)
            wait();
        }
      }
      catch (InterruptedException ex)
```

```
      {
      }
    }
  }

  // Implement the stop method to suspend the thread
  public void stop()
  {
    suspend();
  }

  // Destroy the thread
  public void destroy()
  {
    thread = null;
  }

  // Resume the suspended thread
  public synchronized void resume()
  {
    if (suspended)
    {
      suspended = false;
      notify();
    }
  }

  // Suspend the thread
  public synchronized void suspend()
  {
    suspended = true;
  }
}
```

```
// TestThreads.java: Define threads using the Thread class
public class TestThreads
{
  // Main method
  public static void main(String[] args)
  {
    // Create threads
    PrintChar printA = new PrintChar('a', 100);
    PrintChar printB = new PrintChar('b', 100);
    PrintNum  print100 = new PrintNum(100);

    // Start threads
    print100.start();
    printA.start();
    printB.start();

    // Pause
    System.out.println("Press Ctrl+C to close this window ...");
    MyInput.readInt();
  }
}

// The thread class for printing a specified character
// in specified times
class PrintChar extends Thread
{
  private char charToPrint;                  // The character to print
  private int times;                         // The times to repeat

  // Construct a thread with specified character and number of
  // times to print the character
  public PrintChar(char c, int t)
  {
    charToPrint = c;
    times = t;
  }

  // Vverride the run() method to tell the system
  // what the thread will do
  public void run()
  {
    for (int i=1; i < times; i++)
      System.out.print(charToPrint);
  }
}
```

```
// The thread class for printing number from 1 to n for a given n
class PrintNum extends Thread
{
  private int lastNum;

  // Construct a thread for print 1, 2, ... i
  public PrintNum(int n)
  {
    lastNum = n;
  }

  public void run()
  {
    for (int i=1; i <= lastNum; i++)
      System.out.print(" " + i);
  }
}
```

```
// TestThreadPriority.java: Test thread priorities
public class TestThreadPriority
{
  // Main method
  public static void main(String[] args)
  {
    // Create three threads
    PrintChar printA = new PrintChar('a',200);
    PrintChar printB = new PrintChar('b',200);
    PrintChar printC = new PrintChar('c',200);

    // Set thread priorities
    printA.setPriority(Thread.NORM_PRIORITY);
    printB.setPriority(Thread.NORM_PRIORITY+1);
    printC.setPriority(Thread.NORM_PRIORITY+2);

    // Start threads
    printA.start();
    printB.start();
    printC.start();
  }
}


// TestTransferWithoutSync.java: Demonstrate resource conflict
import Account;
import NegativeAmountException;
import InsufficientFundException;

public class TestTransferWithoutSync
{
  // Main method
  public static void main(String[] args)
  {
    // Determine if all threads are finished
    boolean done = false;

    // Create a savings account with ID 1 and balance 10000
    Account saving = new Account(1, 10000);

    // Create a checking account with ID 2 and balance 0
    Account checking = new Account(2, 0);

    // Create 100 threads in t1 to transfer money from
    // savings to checking
    Thread t1[] = new Thread[100];

    // Create a thread group g1 for grouping t1's
    ThreadGroup g1 = new ThreadGroup("from savings to checking");
```

```
    // Create 100 threads in t2 to transfer money from
    // checking to savings
    Thread t2[] = new Thread[100];

    // Create a thread group g2 for grouping t2's
    ThreadGroup g2 = new ThreadGroup("from checking to savings");

    // Add t1[i] to g1, and start t1[i]
    for (int i=0; i<100; i++)
    {
      t1[i] = new Thread(g1,
        new TransferThread(saving, checking, 1), "t1");
      t1[i].start();
    }

    // Add t2[i] to g2, and start t2[i]
    for (int i=0; i<100; i++)
    {
      t2[i] = new Thread(g2,
        new TransferThread(checking, saving, 1),"t2");
      t2[i].start();
    }

    // Exit the loop when all threads finished
    while (!done)
      if ((g1.activeCount() == 0) && (g2.activeCount() == 0))
        done = true;

    // Show the balance in the savings and checking accounts
    System.out.println("Savings account balance "+
      saving.getBalance());
    System.out.println("Checking account balance "+
      checking.getBalance());

    // Pause
    System.out.println("Press Ctrl+C to close this window ...");
    MyInput.readInt();
  }
}
```

```
// Define the thread to transfer money between accounts
class TransferThread extends Thread
{
  private Account fromAccount, toAccount;
  private double amount;

  // Construct a thread transferring amount
  // from account s to account c
  public TransferThread(Account s, Account c, double amount)
  {
    fromAccount = s;
    toAccount = c;
    this.amount = amount;
  }

  // Override the run method
  public void run()
  {
    transfer(fromAccount, toAccount, amount);
  }

  // Transfer amount from fromAccount to toAccount
  public void transfer(Account fromAccount,
                       Account toAccount,  double amount)
  {
    // Record the balance before transaction for use in recovery
    double fromAccountPriorBalance = fromAccount.getBalance();
    double toAccountPriorBalance = toAccount.getBalance();

    try
    {
      toAccount.deposit(amount);
      sleep(10);
      fromAccount.withdraw(amount);
    }
    catch (NegativeAmountException ex)
    {
      // Reset the balance to the value prior to the exception
      fromAccount.setBalance(fromAccountPriorBalance);
      toAccount.setBalance(toAccountPriorBalance);
    }
    catch (InsufficientFundException ex)
    {
      // Reset the balance to the value prior to the exception
      fromAccount.setBalance(fromAccountPriorBalance);
      toAccount.setBalance(toAccountPriorBalance);
    }
    catch (InterruptedException ex)    {    }
  }
}
```

```
// TestTransferWithSync.java: Demonstrate resource conflict
import Account;
import NegativeAmountException;
import InsufficientFundException;

public class TestTransferWithSync
{
  // Main method
  public static void main(String[] args)
  {
    // Determine if all threads are finished
    boolean done = false;

    // Create a savings account with ID 1 and balance 10000
    Account saving = new Account(1, 10000);

    // Create a checking account with ID 2 and balance 0
    Account checking = new Account(2, 0);

    // Create 100 threads in t1 to transfer money from
    // savings to checking
    Thread t1[] = new Thread[100];

    // Create a thread group g1 for grouping t1's
    ThreadGroup g1 = new ThreadGroup("from savings to checking");

    // Create 100 threads in t2 to transfer money from
    // checking to savings
    Thread t2[] = new Thread[100];

    // Create a thread group g2 for grouping t2's
    ThreadGroup g2 = new ThreadGroup("from checking to savings");

    // Add t1[i] to g1, and start t1[i]
    for (int i=0; i<100; i++)
    {
      t1[i] = new Thread(g1,
        new TransferThread(saving, checking, 1), "t1");
      t1[i].start();
    }

    // Add t2[i] to g2, and start t2[i]
    for (int i=0; i<100; i++)
    {
      t2[i] = new Thread(g2,
        new TransferThread(checking, saving, 1),"t2");
      t2[i].start();
    }
```

```
    // Exit the loop when all threads finished
    while (!done)
      if ((g1.activeCount() == 0) && (g2.activeCount() == 0))
        done = true;

    // Show the balance in the savings and checking accounts
    System.out.println("Savings account balance "+
      saving.getBalance());
    System.out.println("Checking account balance "+
      checking.getBalance());
  }
}

// Define the thread to transfer money between accounts
class TransferThread extends Thread
{
  private Account fromAccount, toAccount;
  private double amount;

  // Construct a thread transferring amount
  // from account s to account c
  public TransferThread(Account s, Account c, double amount)
  {
    fromAccount = s;
    toAccount = c;
    this.amount = amount;
  }

  // Override the run method
  public void run()
  {
    transfer(fromAccount, toAccount, amount);
  }
```

```
  // Transfer amount from fromAccount to toAccount
  public static synchronized void transfer(Account fromAccount,
                    Account toAccount,
                    double amount)
  {
    // Record the balance before transaction for use in recovery
    double fromAccountPriorBalance = fromAccount.getBalance();
    double toAccountPriorBalance = toAccount.getBalance();

    try
    {
      toAccount.deposit(amount);
      sleep(10);
      fromAccount.withdraw(amount);
    }
    catch (NegativeAmountException ex)
    {
      // Reset the balance to the value prior to the exception
      fromAccount.setBalance(fromAccountPriorBalance);
      toAccount.setBalance(toAccountPriorBalance);
    }
    catch (InsufficientFundException ex)
    {
      // Reset the balance to the value prior to the exception
      fromAccount.setBalance(fromAccountPriorBalance);
      toAccount.setBalance(toAccountPriorBalance);
    }
    catch (InterruptedException ex)
    {
    }
  }
}
```