```
// Fig. 12.1: DivideByZeroException.java
// Definition of class DivideByZeroException.
// Used to throw an exception when a
// divide-by-zero is attempted.
public class DivideByZeroException
             extends ArithmeticException {
   public DivideByZeroException()
   {
      super( "Attempted to divide by zero" );
   }
}
```

```
// Fig. 12.1: DivideByZeroTest.java
// A simple exception handling example.
// Checking for a divide-by-zero-error.

import java.io.*;

public class DivideByZeroTest {
  public static void main( String args[] ) {
    BufferedReader in = new BufferedReader(
               new InputStreamReader(System.in) );
    String str1, str2;
    int number1, number2;
    double result;

    try {
      System.out.print("Enter numerator: ");
      number1 = Integer.parseInt( in.readLine() );
      System.out.print("Enter denominator: ");
      number2 = Integer.parseInt( in.readLine() );

      result = quotient( number1, number2 );
      System.out.println( number1 + " / " + number2 + " = " +
                          result );
    }
    catch ( IOException ioe ) {
      System.out.println( "Error reading from Input" );
    }
    catch ( NumberFormatException nfe ) {
      System.out.println( "You must enter two integers" );
    }
    catch ( DivideByZeroException dbze ) {
      System.out.println( dbze.toString() );
    }
  }

  // Definition of method quotient. Used to demonstrate
  // throwing an exception when a divide-by-zero error
  // is encountered.
  public static double quotient( int numerator, int denominator )
       throws DivideByZeroException
  {
    if ( denominator == 0 )
      throw new DivideByZeroException();

    return ( double ) numerator / denominator;
  }
}
```

```
// Fig. 12.8: UsingExceptions.java
// Demonstration of the try-catch-finally
// exception handling mechanism.
public class UsingExceptions {
  public static void main( String args[] )
  {
    try {
      throwException();
    }
    catch ( Exception e )
      {
        System.err.println( "Exception handled in main" );
      }

    doesNotThrowException();
  }

  public static void throwException() throws Exception
  {
    // Throw an exception and immediately catch it.
    try {
      System.out.println( "Method throwException" );
      throw new Exception();  // generate exception
    }
    catch( Exception e )
      {
        System.err.println( "Exception handled in " +
                            "method throwException" );
        throw e;  // rethrow exception for further processing
      }
    finally {
      System.err.println( "Finally is always executed" );
    }
  }

  public static void doesNotThrowException()
  {
    try {
      System.out.println( "Method doesNotThrowException" );
    }
    catch( Exception e )
      {
        System.err.println( e.toString() );
      }
    finally {
      System.err.println( "Finally is always executed." );
    }
  }
```

```
}
```

```
// Fig. 12.9: UsingExceptions.java
// Demonstration of stack unwinding.
public class UsingExceptions {
  public static void main( String args[] )
  {
    try {
      throwException();
    }
    catch ( Exception e )
      {
        System.err.println( "Exception handled in main" );
      }
  }

  public static void throwException() throws Exception
  {
    // Throw an exception and catch it in main.
    try {
      System.out.println( "Method throwException" );
      throw new Exception();  // generate exception
    }
    catch( OtherException e )
      {
        System.err.println( "Exception handled in " +
                            "method throwException" );
      }
    finally {
      System.err.println( "Finally is always executed" );
    }
  }
}

class OtherException extends Exception {
  public OtherException()
  {
    super( "Another exception type" );
  }
}
```

```java
// Fig. 12.10: UsingExceptions.java
// Demonstrating the getMessage and printStackTrace
// methods inherited into all exception classes.
public class UsingExceptions {
  public static void main( String args[] )
  {
    try {
      method1();
    }
    catch ( UserException e )
      {
        System.err.println( e.getMessage() +
                            "\nThe stack trace is:" );
        e.printStackTrace();
      }
  }

  public static void method1() throws UserException
  {
    method2();
  }

  public static void method2() throws UserException
  {
    method3();
  }

  public static void method3() throws UserException
  {
    throw new UserException();
  }
}

class UserException extends Exception {
  public UserException()
  {
    super( "This is a user defined exception." );
  }
}
```

```
// Fig. 13.3: PrintTest.java
// Show multiple threads printing at different intervals.

public class PrintTest {
  public static void main( String args[] )
  {
    PrintThread thread1, thread2, thread3, thread4;

    thread1 = new PrintThread();
    thread2 = new PrintThread();
    thread3 = new PrintThread();
    thread4 = new PrintThread();

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();
  }
}

class PrintThread extends Thread {
  int sleepTime;

  // PrintThread constructor assigns name to thread
  // by calling Thread constructor
  public PrintThread()
  {
    // sleep between 0 and 5 seconds
    sleepTime = (int) ( Math.random() * 5000 );

    System.out.println( "Name: " + getName() +
                        ";  sleep: " + sleepTime );
  }

  // execute the thread
  public void run()
  {
    // put thread to sleep for a random interval
    try {
      Thread.sleep( sleepTime );
    }
    catch ( InterruptedException exception ) {
      System.err.println( exception.toString() );
    }

    // print thread name
    System.out.println( getName() );
  }
```

```
}
```

```java
// Fig. 13.4: SharedCell.java
// Show multiple threads modifying shared object.
// Illustrate how things fail without synchronization
// DO NOT BASE YOUR OWN CODE ON THIS FILE -- It is broken!

public class SharedCell {
  public static void main( String args[] )
  {
    HoldInteger h = new HoldInteger();
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );

    p.start();
    c.start();
  }
}

class ProduceInteger extends Thread {
  private HoldInteger pHold;

  public ProduceInteger( HoldInteger h )
  {
    pHold = h;
  }

  public void run()
  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      pHold.setSharedInt( count );
      System.out.println( "Producer set sharedInt to " +
                          count );
    }

    pHold.setMoreData( false );
  }
}

class ConsumeInteger extends Thread {
  private HoldInteger cHold;
```

```java
  public ConsumeInteger( HoldInteger h )
  {
    cHold = h;
  }

  public void run()
  {
    int val;

    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      val = cHold.getSharedInt();
      System.out.println( "Consumer retrieved " + val );
    }
  }
}

class HoldInteger {
  private int sharedInt = -1;
  private boolean moreData = true;

  public void setSharedInt( int val ) { sharedInt = val; }

  public int getSharedInt() { return sharedInt; }

  public void setMoreData( boolean b ) { moreData = b; }

  public boolean hasMoreData() { return moreData; }
}
```

```java
// Fig. 13.5: SharedCell.java
// Show multiple threads modifying shared object.
// Use synchronization to ensure that both threads
// access the shared cell properly.

public class SharedCell {
  public static void main( String args[] )
  {
    HoldInteger h = new HoldInteger();
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );

    p.start();
    c.start();
  }
}

class ProduceInteger extends Thread {
  private HoldInteger pHold;

  public ProduceInteger( HoldInteger h )
  {
    pHold = h;
  }

  public void run()
  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      pHold.setSharedInt( count );
      System.out.println( "Producer set sharedInt to " +
                          count );
    }

    pHold.setMoreData( false );
  }
}

class ConsumeInteger extends Thread {
  private HoldInteger cHold;
```

```
  public ConsumeInteger( HoldInteger h )
  {
    cHold = h;
  }

  public void run()
  {
    int val;

    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      val = cHold.getSharedInt();
      System.out.println( "Consumer retrieved " + val );
    }
  }
}

class HoldInteger {
  private int sharedInt = -1;
  private boolean moreData = true;
  private boolean writeable = true;

  public synchronized void setSharedInt( int val )
  {
    while ( !writeable ) {
      try {
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( "Exception: " + e.toString() );
      }
    }

    sharedInt = val;
    writeable = false;
    notify();
  }

  public synchronized int getSharedInt()
  {
    while ( writeable ) {
```

```
      try {
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( "Exception: " + e.toString() );
      }
    }

    writeable = true;
    notify();
    return sharedInt;
  }

  public void setMoreData( boolean b ) { moreData = b; }

  public boolean hasMoreData() { return moreData; }
}
```

```
// Fig. 13.6: SharedCell.java
// Show multiple threads modifying shared object.
// Use synchronization to ensure that both threads
// access the shared cell properly.
import java.text.DecimalFormat;

public class SharedCell {
  public static void main( String args[] )
  {
    HoldInteger h = new HoldInteger();
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );

    p.start();
    c.start();
  }
}

class ProduceInteger extends Thread {
  private HoldInteger pHold;

  public ProduceInteger( HoldInteger h )
  {
    pHold = h;
  }

  public void run()
  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 500 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      pHold.setSharedInt( count );
      // System.out.println( "Producer set sharedInt to " +
      //                     count );
    }

    pHold.setMoreData( false );
  }
}

class ConsumeInteger extends Thread {
  private HoldInteger cHold;
```

```
  public ConsumeInteger( HoldInteger h )
  {
    cHold = h;
  }

  public void run()
  {
    int val;

    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }

      val = cHold.getSharedInt();
      // System.out.println( "Consumer retrieved " + val );
    }
    System.out.println();
  }
}

class HoldInteger {
  private int sharedInt[] = { -1, -1, -1, -1, -1 };
  private boolean moreData = true;
  private boolean writeable = true;
  private boolean readable = false;
  private int readLoc = 0, writeLoc = 0;

  public synchronized void setSharedInt( int val )
  {
    while ( !writeable ) {
      try {
        System.out.print( " WAITING TO PRODUCE " + val );
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( e.toString() );
      }
    }

    sharedInt[ writeLoc ] = val;
    readable = true;
```

```
    System.out.print( "\nProduced " + val +
                      " into cell " + writeLoc );

    writeLoc = ( writeLoc + 1 ) % 5;

    System.out.print( "\twrite " + writeLoc +
                      "\tread " + readLoc );
    printBuffer( sharedInt );

    if ( writeLoc == readLoc ) {
      writeable = false;
      System.out.print( "\nBUFFER FULL" );
    }

    notify();
  }

  public synchronized int getSharedInt()
  {
    int val;

    while ( !readable ) {
      try {
        System.out.print( " WAITING TO CONSUME" );
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( e.toString() );
      }
    }

    writeable = true;
    val = sharedInt[ readLoc ];

    System.out.print( "\nConsumed " + val +
                      " from cell " + readLoc );

    readLoc = ( readLoc + 1 ) % 5;

    System.out.print( "\twrite " + writeLoc +
                      "\tread " + readLoc );
    printBuffer( sharedInt );

    if ( readLoc == writeLoc ) {
      readable = false;
      System.out.print( "\nBUFFER EMPTY" );
    }
```

```
    notify();
    return val;
  }

  public void printBuffer( int buf[] )
  {
    DecimalFormat threeChars = new DecimalFormat( " #;-#" );
    System.out.print( "\tbuffer: " );

    for ( int i = 0; i < buf.length; i++ )
      System.out.print( " " + threeChars.format( buf[ i ] ) );
  }

  public void setMoreData( boolean b ) { moreData = b; }

  public boolean hasMoreData()
  {
    if ( moreData == false && readLoc == writeLoc )
      return false;
    else
      return true;
  }
}
```