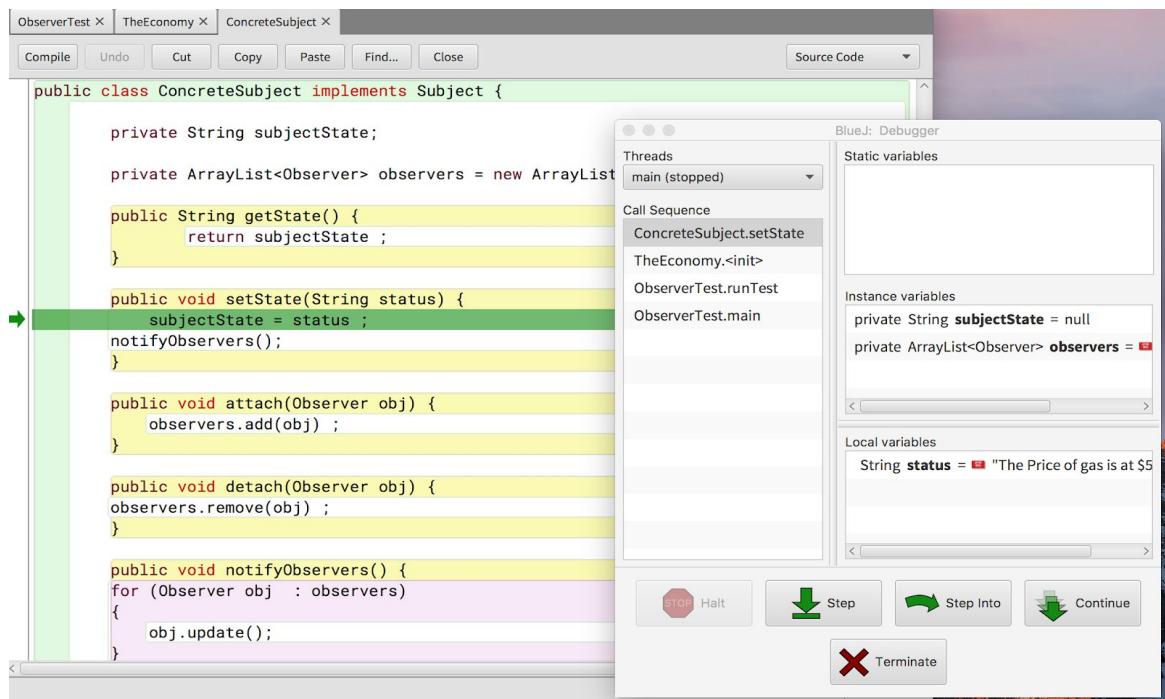
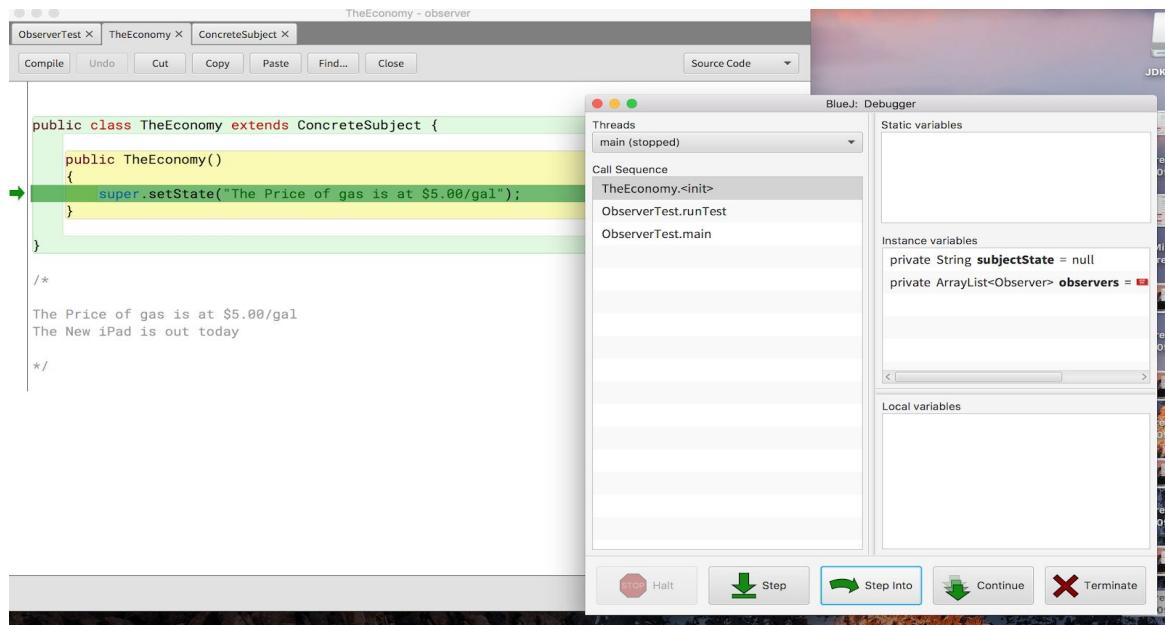


Lab 3: Observer Pattern runtime screenshots

Step1: setState(status: String) : void



Step 1.1: notifyObservers() : void

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the file 'ConcreteSubject - observer' open. The code implements the Subject interface with methods for getting and setting the subject state, attaching and detaching observers, and notifying observers. A green arrow points to the 'notifyObservers()' method. On the right is the 'BlueJ: Debugger' window, which displays the current thread (main), call sequence, static variables, instance variables, and local variables. The local variable 'status' is set to "The Price of gas is at \$5".

```
public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public String getState() {
        return subjectState;
    }
    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }
    public void attach(Observer obj) {
        observers.add(obj);
    }
    public void detach(Observer obj) {
        observers.remove(obj);
    }
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}
```

This screenshot is similar to the one above, but the code has been modified. The 'notifyObservers()' method now contains a loop that iterates over all observers and calls their 'update()' method. A green arrow points to the start of this loop. The debugger window shows the same state as before, with the local variable 'status' set to "The Price of gas is at \$5".

```
public class ConcreteSubject implements Subject {
    private String subjectState;
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public String getState() {
        return subjectState;
    }
    public void setState(String status) {
        subjectState = status;
        notifyObservers();
    }
    public void attach(Observer obj) {
        observers.add(obj);
    }
    public void detach(Observer obj) {
        observers.remove(obj);
    }
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }
}
```

Step 2: attach(obj: Observer) : void

The screenshot shows the BlueJ IDE interface. The left pane displays the `ObserverTest.java` source code. A red arrow points to the line `s.attach(p);`. The right pane is the "BlueJ: Debugger" window, which includes tabs for "Threads", "Call Sequence", "Static variables", "Instance variables", and "Local variables". The "Threads" tab shows "main (stopped)". The "Call Sequence" tab shows the stack: `ObserverTest.runTest`, `ObserverTest.main`. The "Local variables" tab shows references to `TheEconomy s`, `Pessimist p`, and `Optimist o`. At the bottom are buttons for "STOP Halt", "Step", "Step Into" (highlighted in blue), and "Continue". A note at the bottom of the code editor says "Cannot set breakpoint no code in this line."

```

public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest() ;
        t.runTest() ;
    }
}

```

This screenshot shows the BlueJ IDE with the `ConcreteSubject.java` code in the left pane. A green arrow points to the `attach` method. The right pane is the "BlueJ: Debugger" window. The "Call Sequence" tab shows the stack: `ConcreteSubject.attach`, `ObserverTest.runTest`, `ObserverTest.main`. The "Local variables" tab shows `subjectState` and `observers`. A red callout box labeled "Pessimist" appears over the "Inspect" button, containing the code for the `Pessimist` object's `update` method. The "Show static fields" button is also visible.

```

private String subjectState;
private ArrayList<Observer> observers = new ArrayList<>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}

```

Step 3: attach(obj: Observer) : void

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the ObserverTest class. A breakpoint is set at the first line of the runTest() method. The code is as follows:

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest() ;
        t.runTest() ;
    }
}
```

The debugger window on the right shows the current state of the application. The threads list shows "main (stopped)". The call sequence shows "ObserverTest.runTest" and "ObserverTest.main". The local variables pane shows references to TheEconomy s, Pessimist p, and Optimist o. A tooltip for s indicates it is an Optimist object.

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the ConcreteSubject class. A breakpoint is set at the attach() method. The code is as follows:

```
private String subjectState;

private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

The debugger window on the right shows the current state of the application. The threads list shows "main (stopped)". The call sequence shows "ConcreteSubject.attach", "ObserverTest.runTest", and "ObserverTest.main". The local variables pane shows a reference to an Observer obj. A tooltip for obj indicates it is an Optimist object. A detailed view of the Optimist object is shown in a red-bordered window, displaying its protected fields: observerState (null) and ConcreteSubject s... (with a circular arrow icon).

Step 4: setState(status: String) : void

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the file `ObserverTest.java`. The code is as follows:

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest() ;
        t.runTest() ;
    }
}
```

A red arrow points to the line `s.setState("The New iPad is out today");`, indicating the current execution point. The right side of the interface is the BlueJ Debugger window, which displays the call stack, local variables, and instance variables.

Call Sequence:

- ObserverTest.runTest
- ObserverTest.main

Local variables:

- TheEconomy s = <object reference>
- Pessimist p = <object reference>
- Optimist o = <object reference>

Buttons at the bottom of the debugger window include: STOP Halt, Step, Step Into, Continue, and Terminate.

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the file `ConcreteSubject.java`. The code is as follows:

```
private String subjectState;
private ArrayList<Observer> observers = new ArrayList<>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

A green arrow points to the line `subjectState = status ;`, indicating the current execution point. The right side of the interface is the BlueJ Debugger window, which displays the call stack, local variables, and instance variables.

Call Sequence:

- ConcreteSubject.setState
- ObserverTest.runTest
- ObserverTest.main

Local variables:

- String status = "The New iPad is out today"

Buttons at the bottom of the debugger window include: STOP Halt, Step, Step Into, Continue, and Terminate.

Step 4.1: notifyObservers() : void

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the ConcreteSubject class. A green arrow points to the `notifyObservers()` method. The code is as follows:

```
private String subjectState;
private ArrayList<Observer> observers = new ArrayList<>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

On the right is the BlueJ Debugger window. It shows the call stack: `ConcreteSubject.setState`, `ObserverTest.runTest`, and `ObserverTest.main`. The current thread is `main (stopped)`. The static variables pane shows `subjectState` and `observers`. The instance variables pane shows `status` with the value "The New iPad is out today". The local variables pane is empty. At the bottom are buttons for `Stop Halt`, `Step`, `Step Into`, `Continue`, and `Terminate`.

This screenshot is identical to the one above, showing the same code in the ConcreteSubject class and the same state in the BlueJ Debugger window. The green arrow points to the `notifyObservers()` method, which is currently executing.

Step 4.1.1: update() : void

The screenshot shows the BlueJ IDE interface. The current class is `ConcreteSubject`. A breakpoint is set on the line `obj.update();`. A tooltip for the `Pessimist` object shows its state:

```

: Pessimist
protected String observerState null
protected ConcreteSubject s...

```

The screenshot shows the BlueJ IDE interface. The current class is `Pessimist`. A breakpoint is set on the first line of the `update()` method. A tooltip for the `Pessimist` object shows its state:

```

: Pessimist
protected String observerState null
protected ConcreteSubject subject ...

```

Step 4.1.1.1: getState() : String

The screenshot shows the BlueJ IDE interface. On the left is the code editor with the `ObserverTest` class open. The current line of code is highlighted in green, showing the implementation of the `getState()` method. The code editor includes standard toolbar buttons for Compile, Undo, Cut, Copy, Paste, Find..., and Close. To the right is the BlueJ Debugger window, which has several tabs: Threads, Call Sequence, Static variables, Instance variables, and Local variables. The Threads tab shows the main thread is stopped. The Call Sequence tab shows the call stack from `ConcreteSubject.getState()` down to `ObserverTest.main`. The Instance variables tab shows the `subjectState` field is set to "The New iPa". The Local variables tab is currently empty. At the bottom of the debugger are control buttons: STOP Halt, Step, Step Into (which is highlighted in blue), Continue, and Terminate.

```
private String subjectState;

private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

BlueJ: Debugger

Threads
main (stopped)

Call Sequence
ConcreteSubject.getState
Pessimist.update
ConcreteSubject.notifyO...
ConcreteSubject.setState
ObserverTest.runTest
ObserverTest.main

Static variables

Instance variables
private String **subjectState** = "The New iPa";
private ArrayList<Observer> **observers** = <empty>

Local variables

STOP Halt Step Step Into Continue Terminate

Step 4.1.2: update() : void

The screenshot shows the BlueJ IDE interface. The code editor displays the `ConcreteSubject` class with its `update()` method highlighted. The debugger window shows the `main` thread stopped at the `obj.update()` call. The `Optimist` object's local variables are shown: `observerState` is `null` and `subject` is a reference to a `ConcreteSubject` object.

```

private String subjectState;
private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}

```

The screenshot shows the BlueJ IDE interface. The code editor displays the `Optimist` class, which extends `ConcreteObserver`. The `update()` method is highlighted. The debugger window shows the `main` thread stopped at the `obj.update()` call. The `Optimist` object's local variables are shown: `observerState` is `null` and `subject` is a reference to a `ConcreteSubject` object.

```

public class Optimist extends ConcreteObserver {
    public Optimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Pr...") )
        {
            observerState = "Great! It's time to go green."
        }
        else if ( subject.getState().equalsIgnoreCase( "The N...") )
        {
            observerState = "Apple, take my money!" ;
        }
        else
        {
            observerState = ":)" ;
        }
    }
}

```

Step 4.1.2.1: getState() : String

The screenshot shows the BlueJ IDE interface. The top menu bar includes tabs for ObserverTest, TheEconomy, ConcreteSubject, Pessimist, ConcreteObserver, and Optimist. Below the menu is a toolbar with Compile, Undo, Cut, Copy, Paste, Find..., and Close buttons. The main window displays the source code of the ConcreteSubject class:

```
private String subjectState;  
private ArrayList<Observer> observers = new ArrayList<Observer>();  
  
public String getState() {  
    return subjectState;  
}  
  
public void setState(String status) {  
    subjectState = status;  
    notifyObservers();  
}  
  
public void attach(Observer obj) {  
    observers.add(obj);  
}  
  
public void detach(Observer obj) {  
    observers.remove(obj);  
}  
  
public void notifyObservers() {  
    for (Observer obj : observers)  
    {  
        obj.update();  
    }  
}
```

The code editor highlights the `getState()` method. To the right is the BlueJ Debugger window, which shows the following details:

- Threads:** main (stopped)
- Call Sequence:** ConcreteSubject.getState → Optimist.update → ConcreteSubject.notifyObservers → ConcreteSubject.setState → ObserverTest.runTest → ObserverTest.main
- Static variables:** None
- Instance variables:** `private String subjectState = "The New iPad";`, `private ArrayList<Observer> observers = <empty>`
- Local variables:** None

At the bottom of the debugger window are control buttons: STOP Halt, Step, Step Into, Continue, and Terminate.

Step 5: showState() : void

The screenshot shows the BlueJ IDE interface. The source code editor displays the `ObserverTest` class. A red stop sign icon is placed before the line `p.showState();`, indicating a breakpoint. The code is as follows:

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest();
        t.runTest();
    }
}
```

The BlueJ Debugger window is open, showing the call sequence: `ObserverTest.runTest` and `ObserverTest.main`. The local variables pane shows references to `s`, `p`, and `o`. The control buttons at the bottom include `STOP Halt`, `Step`, `Step Into` (highlighted in blue), and `Continue`.

The screenshot shows the BlueJ IDE interface. The source code editor displays the `ConcreteObserver` class, which implements the `Observer` interface. A green arrow icon is placed before the line `System.out.println("Observer: " + this.getClass().get`, indicating a breakpoint. The code is as follows:

```
public class ConcreteObserver implements Observer
{
    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject;
    }

    public void update()
    {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().get
    }
}
```

The BlueJ Debugger window is open, showing the call sequence: `ConcreteObserver.showS`, `ObserverTest.runTest`, and `ObserverTest.main`. The local variables pane shows `observerState` and `subject`. The control buttons at the bottom include `STOP Halt`, `Step`, `Step Into` (highlighted in blue), and `Continue`.

ObserverTest X TheEconomy X ConcreteSubject X Pessimist X ConcreteObserver X Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code

Search

BlueJ: Debugger

Threads main (stopped)

Call Sequence Pessimist.update

ConcreteSubject.notifyObservers

ConcreteSubject.setState

ObserverTest.runTest

ObserverTest.main

Instance variables

protected String observerState = null

protected ConcreteSubject subject = null <object>

Local variables

Step Halt Step Into Continue Terminate

```
public class Pessimist extends ConcreteObserver {
    public Pessimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price is Right" ) )
        {
            observerState = "This is the beginning of the end";
        }
        else if ( subject.getState().equalsIgnoreCase( "The Next Big Thing" ) )
        {
            observerState = "Not another iPad!";
        }
        else
        {
            observerState = ":(" ;
        }
    }
}
```

Step 6: showState() : void

ObserverTest X TheEconomy X ConcreteSubject X Pessimist X ConcreteObserver X Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code Diagram

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest();
        t.runTest();
    }
}
```

Cannot set breakpoint: no code in this line.

BlueJ: Debugger

Threads: main (stopped)

Call Sequence: ObserverTest.runTest, ObserverTest.main

Static variables:

- TheEconomy s = <object reference>
- Pessimist p = <object reference>
- Optimist o = <object reference>

Instance variables:

Local variables:

Control buttons: STOP Halt, Step, Step Into, Continue, Terminate

ObserverTest X TheEconomy X ConcreteSubject X Pessimist X ConcreteObserver X Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code Search

```
public class ConcreteObserver implements Observer {
    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject ;
    }

    public void update()
    {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().get
    }
}
```

BlueJ: Debugger

Threads: main (stopped)

Call Sequence: ConcreteObserver.showS, ObserverTest.runTest, ObserverTest.main

Static variables:

Instance variables:

Local variables:

Control buttons: STOP Halt, Step, Step Into, Continue, Terminate

ObserverTest X | TheEconomy X | ConcreteSubject X | Pessimist X | ConcreteObserver X | Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code

```
public class Optimist extends ConcreteObserver {
    public Optimist( ConcreteSubject sub )
    {
        super( sub );
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Price Is Right") )
        {
            observerState = "Great! It's time to go green."
        }
        else if ( subject.getState().equalsIgnoreCase( "The Next Big Thing" ) )
        {
            observerState = "Apple, take my money!";
        }
        else
        {
            observerState = ":)";
        }
    }
}
```

BlueJ: Debugger

Threads main (stopped)

Call Sequence Optimist.update

ConcreteSubject.notifyObservers

ConcreteSubject.setState

ObserverTest.runTest

ObserverTest.main

Static variables

Instance variables

protected String **observerState** = null

protected ConcreteSubject **subject** =  <object>

Local variables

STOP Halt Step Step Into Continue Terminate

Step 7: setState(status: String) : void

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the file `ObserverTest.java`. The code is as follows:

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest();
        t.runTest();
    }
}
```

The line `s.setState("Hey, Its Friday!");` is highlighted in green and has a green arrow pointing to it from the left margin, indicating it is the next line to be executed. The right side of the interface is the BlueJ Debugger window, which includes tabs for Threads, Call Sequence, Static variables, Instance variables, and Local variables. The Call Sequence tab shows the current call stack: `ObserverTest.runTest` and `ObserverTest.main`. The Local variables section shows references to `TheEconomy s`, `Pessimist p`, and `Optimist o`. At the bottom are standard debugger controls: STOP Halt, Step, Step Into (which is highlighted in blue), Continue, and Terminate.

The screenshot shows the BlueJ IDE interface. On the left is the source code editor with the file `ConcreteSubject.java`. The code is as follows:

```
private String subjectState;

private ArrayList<Observer> observers = new ArrayList<>();

public String getState() {
    return subjectState;
}

public void setState(String status) {
    subjectState = status;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj);
}

public void detach(Observer obj) {
    observers.remove(obj);
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

The line `subjectState = status;` is highlighted in green and has a green arrow pointing to it from the left margin, indicating it is the next line to be executed. The right side of the interface is the BlueJ Debugger window, which includes tabs for Threads, Call Sequence, Static variables, Instance variables, and Local variables. The Call Sequence tab shows the current call stack: `ConcreteSubject.setState`, `ObserverTest.runTest`, and `ObserverTest.main`. The Local variables section shows the variable `status` with the value "Hey, Its Friday!". At the bottom are standard debugger controls: STOP Halt, Step, Step Into (highlighted in blue), Continue, and Terminate.

Step 7.1: notifyObservers() : void

The screenshot shows the BlueJ IDE interface. The source code editor displays the `ObserverTest` class. The `notifyObservers()` method is highlighted with a green selection bar. The BlueJ debugger window is open, showing the call stack: `main (stopped)`, `ConcreteSubject.setState`, `ObserverTest.runTest`, and `ObserverTest.main`. The local variables pane shows a `String status` variable with the value "Hey, Its Friday!". The control buttons at the bottom include `STOP Halt`, `Step`, `Step Into`, `Continue`, and `Terminate`.

```
private String subjectState;
private ArrayList<Observer> observers = new ArrayList<Observer>();
public String getState() {
    return subjectState ;
}
public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}
public void attach(Observer obj) {
    observers.add(obj) ;
}
public void detach(Observer obj) {
    observers.remove(obj) ;
}
public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

The screenshot shows the BlueJ IDE interface. The source code editor displays the `ConcreteSubject` class. The `notifyObservers()` method is highlighted with a green selection bar. The BlueJ debugger window shows the call stack: `main (stopped)`, `ConcreteSubject.notifyObservers`, `ConcreteSubject.setState`, `ObserverTest.runTest`, and `ObserverTest.main`. The local variables pane is empty. The control buttons at the bottom include `STOP Halt`, `Step`, `Step Into`, `Continue`, and `Terminate`.

```
private String subjectState;
private ArrayList<Observer> observers = new ArrayList<Observer>();
public String getState() {
    return subjectState ;
}
public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}
public void attach(Observer obj) {
    observers.add(obj) ;
}
public void detach(Observer obj) {
    observers.remove(obj) ;
}
public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

Step 7.1.1: update() : void

The screenshot shows the BlueJ IDE interface. The code editor displays the `ConcreteSubject` class with its `update()` method highlighted. The debugger window on the right shows the stack trace: `main (stopped)`, `ConcreteSubject.notifyObservers()`, `ConcreteSubject.setState()`, `ObserverTest.runTest`, and `ObserverTest.main`. A red callout box points to the `: Pessimist` entry in the stack trace, indicating the current thread. The local variables pane shows an `Observer obj` reference.

```
private String subjectState;
private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

The screenshot shows the BlueJ IDE interface. The code editor displays the `Pessimist` class, which extends `ConcreteObserver`. The `update()` method is highlighted. The debugger window shows the stack trace: `main (stopped)`, `Pessimist.update()`, `ConcreteSubject.notifyObservers()`, `ConcreteSubject.setState()`, `ObserverTest.runTest`, and `ObserverTest.main`. A red callout box points to the `Pessimist.update()` entry in the stack trace. The local variables pane shows the `observerState` variable containing the value "Not another iPad!" and a reference to the `ConcreteSubject` object.

```
public class Pessimist extends ConcreteObserver {

    public Pessimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update()
    {
        if ( subject.getState().equalsIgnoreCase("The Price is too high" ) )
        {
            observerState = "This is the beginning of the end";
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is here!" ) )
        {
            observerState = "Not another iPad!";
        }
        else
        {
            observerState = ":(" ;
        }
    }
}
```

Step 7.1.1.1: getState() : String

The screenshot shows the BlueJ IDE interface with the following components:

- Top Bar:** Shows tabs for ObserverTest, TheEconomy, ConcreteSubject, Pessimist, ConcreteObserver, and Optimist. The ConcreteSubject tab is active.
- Toolbar:** Includes buttons for Compile, Undo, Cut, Copy, Paste, Find..., and Close.
- Source Code Editor:** Displays the code for the ConcreteSubject class. The current line of code, "return subjectState ;", is highlighted in green, indicating it is being executed.
- BlueJ Debugger:** A floating window showing the state of the application.
 - Threads:** Shows "main (stopped)".
 - Call Sequence:** Shows the call stack: ConcreteSubject.getState → Pessimist.update → ConcreteSubject.notifyObservers → ObserverTest.runTest → ObserverTest.main.
 - Instance variables:** Shows the value of `subjectState` as "Hey, Its Friday!" and the `observers` list as <object reference>.
 - Local variables:** Currently empty.
- Bottom Buttons:** Includes STOP Halt, Step, Step Into, Continue, and Terminate.

Step 7.1.2: update() : void

```
private String subjectState;

private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

Step 7.1.2.1: getState() : String

```
private String subjectState;

private ArrayList<Observer> observers = new ArrayList<Observer>();

public String getState() {
    return subjectState ;
}

public void setState(String status) {
    subjectState = status ;
    notifyObservers();
}

public void attach(Observer obj) {
    observers.add(obj) ;
}

public void detach(Observer obj) {
    observers.remove(obj) ;
}

public void notifyObservers() {
    for (Observer obj : observers)
    {
        obj.update();
    }
}
```

Step 8: showState() : void

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest();
        t.runTest();
    }
}
```

```
public class ConcreteObserver implements Observer {
    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject;
    }

    public void update()
    {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().get...
    }
}
```

ObserverTest X | TheEconomy X | ConcreteSubject X | Pessimist X | ConcreteObserver X | Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code Search

```
public class Pessimist extends ConcreteObserver {  
    public Pessimist( ConcreteSubject sub )  
    {  
        super( sub ) ;  
    }  
  
    public void update() {  
        if ( subject.getState().equalsIgnoreCase("The Price is up") )  
        {  
            observerState = "This is the beginning of the end." ;  
        }  
        else if ( subject.getState().equalsIgnoreCase( "The News is bad" ) )  
        {  
            observerState = "Not another iPad!" ;  
        }  
        else  
        {  
            observerState = ":(" ;  
        }  
    }  
}
```

BlueJ: Debugger

Threads main (stopped)

Call Sequence Pessimist.update

ConcreteSubject.notifyObservers()
ConcreteSubject.setState()
ObserverTest.runTest
ObserverTest.main

Static variables

Instance variables

ring **observerState** = "Not another iPad!"
ConcreteSubject **subject** = <object reference>

Local variables

STOP Halt Step Step Into Continue Terminate

Step 9: showState() : void

The screenshot shows the BlueJ IDE interface. The source code editor displays `ObserverTest.java`. A breakpoint is set at the start of the `runTest()` method. The code is as follows:

```
public class ObserverTest
{
    public void runTest()
    {
        TheEconomy s = new TheEconomy();
        Pessimist p = new Pessimist(s);
        Optimist o = new Optimist(s);
        s.attach(p);
        s.attach(o);
        s.setState("The New iPad is out today");
        p.showState();
        o.showState();
        s.setState("Hey, Its Friday!");
        p.showState();
        o.showState();
    }

    public static void main( String [] args )
    {
        ObserverTest t = new ObserverTest();
        t.runTest();
    }
}
```

The debugger sidebar shows the current state:

- Threads:** main (stopped)
- Call Sequence:** ObserverTest.runTest → ObserverTest.main
- Static variables:** None
- Instance variables:** None
- Local variables:**
 - TheEconomy s = [object reference]
 - Pessimist p = [object reference]
 - Optimist o = [object reference]

At the bottom, there are buttons for **STOP Halt**, **Step**, **Step Into** (highlighted), **Continue**, and **Terminate**.

The screenshot shows the BlueJ IDE interface. The source code editor displays `ConcreteObserver.java`. A breakpoint is set at the start of the `showState()` method. The code is as follows:

```
public class ConcreteObserver implements Observer
{
    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject;
    }

    public void update() {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().get
    }
}
```

The debugger sidebar shows the current state:

- Threads:** main (stopped)
- Call Sequence:** ConcreteObserver.showS → ObserverTest.runTest → ObserverTest.main
- Static variables:** None
- Instance variables:**
 - protected String **observerState** = ":"
 - protected ConcreteSubject **subject** = [object reference]
- Local variables:** None

At the bottom, there are buttons for **STOP Halt**, **Step**, **Step Into**, **Continue**, and **Terminate**.

ObserverTest X | TheEconomy X | ConcreteSubject X | Pessimist X | ConcreteObserver X | Optimist X

Compile Undo Cut Copy Paste Find... Close Source Code

BlueJ: Debugger

```
public class Optimist extends ConcreteObserver {  
    public Optimist( ConcreteSubject sub )  
    {  
        super( sub ) ;  
  
        public void update() {  
            if ( subject.getState().equalsIgnoreCase("The Price Is Right") )  
            {  
                observerState = "Great! It's time to go green." ;  
            }  
            else if ( subject.getState().equalsIgnoreCase( "The Next Big Thing" ) )  
            {  
                observerState = "Apple, take my money!" ;  
            }  
            else  
            {  
                observerState = ":)" ;  
            }  
        }  
    }  
}
```

Threads
main (stopped)

Call Sequence
Optimist.update
ConcreteSubject.notifyObservers
ConcreteSubject.setState
ObserverTest.runTest
ObserverTest.main

Static variables

Instance variables
protected String **observerState** = "Apple, take my money!" ;
protected ConcreteSubject **subject** = <object>

Local variables

Step Halt Step Into Continue Terminate