Some Miscellaneous Items

Static, Javadoc, Debugger, Compound Stmts

Produced by: Dr. Siobhán Drohan

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Typical compilation (syntax) errors

```
public class CokeMachine
{
   private price;

   public CokeMachine()
   {
     price = 300
   }

   public int getPrice
   {
     return Price;
   }
```

What is wrong here? Can you spot the 5 errors?

Typical compilation (syntax) errors

```
public class CokeMachine
{
                int
   private price;

   public CokeMachine()
   {
     price = 300;
   }

   public int getPrice()
   {
     return _Price;
   }
}
```

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Default constructor

- A default constructor has NO parameters e.g.:

```
public Square()
```

- If no constructor is defined in a class, java automatically defines a default constructor (note: it is not visible in your written code).

Multiple constructors

- A class can have multiple constructors.
- To have multiple constructors, the parameter list for each constructor must be different.
- Multiple constructors allow you to initialise an object in several different ways.

Valid constructors in a Square class

```
public Square (int height)

public Square (int height, int width)

public Square (int height, int width,
               String colour)

public Square (String colour,
               int xPosition,
               int yPosition)
```

Invalid constructors in a Square class

```
public Square (int height)

public Square (int width)   /* this one is invalid as a
constructor with a single int parameter has already been
defined above.  The name of the parameter is different,
but Java doesn't look at this…it is only interested in the
type.*/

public Square (String colour, int height, int width)

public Square (String colour, int xPosition,
                   int yPosition)
/* this one is invalid as a constructor with a parameter
list of (String, int, int) has already been defined just
above it.*/
```

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

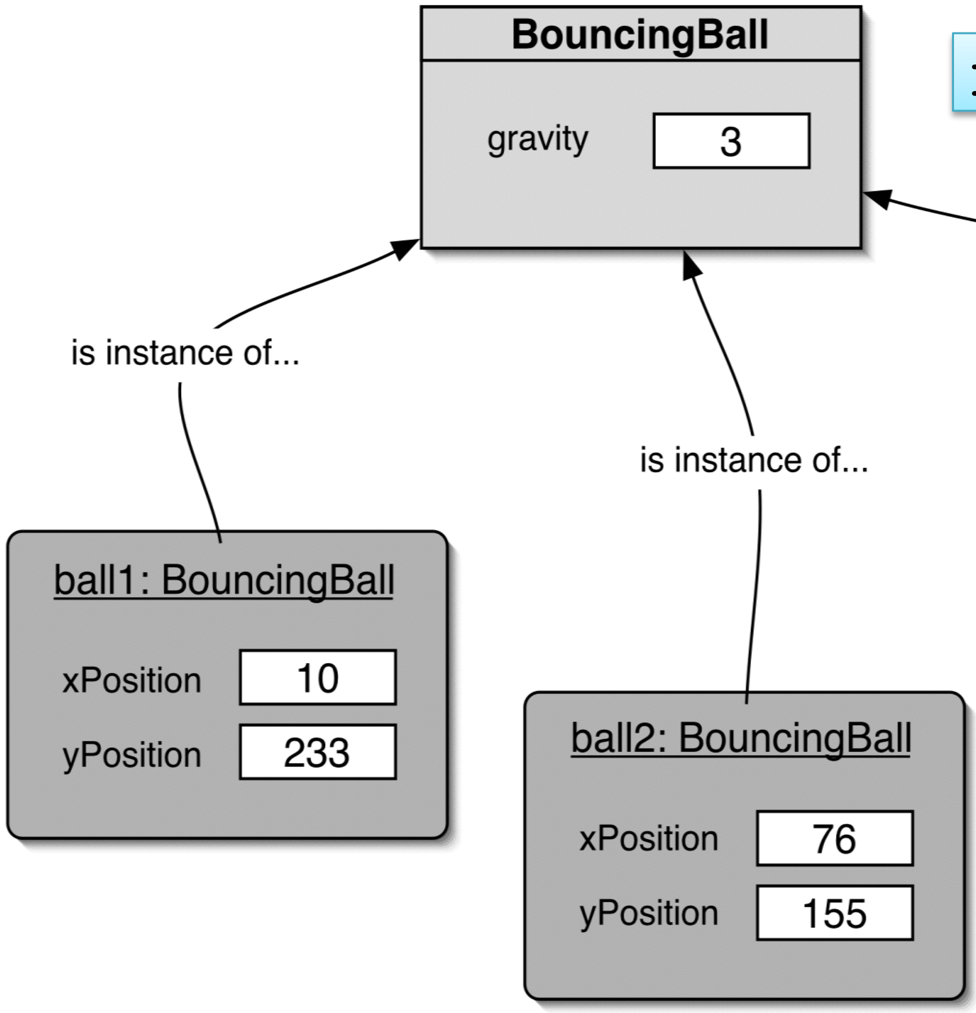
Instance vs Static (Class) Variables

- When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*.

- Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier.
- Fields that have the static modifier in their declaration are called *static fields* or *class variables*.

https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Instance vs Static (Class) Variables



```
private static int gravity;
```

Constants

```
private static final int GRAVITY = 3;
```

- **`private`**: access modifier, as usual
- **`static`**: class variable
- **`final`**: constant (cannot change the value). Naming standards for final fields is all capitals.

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Static Methods

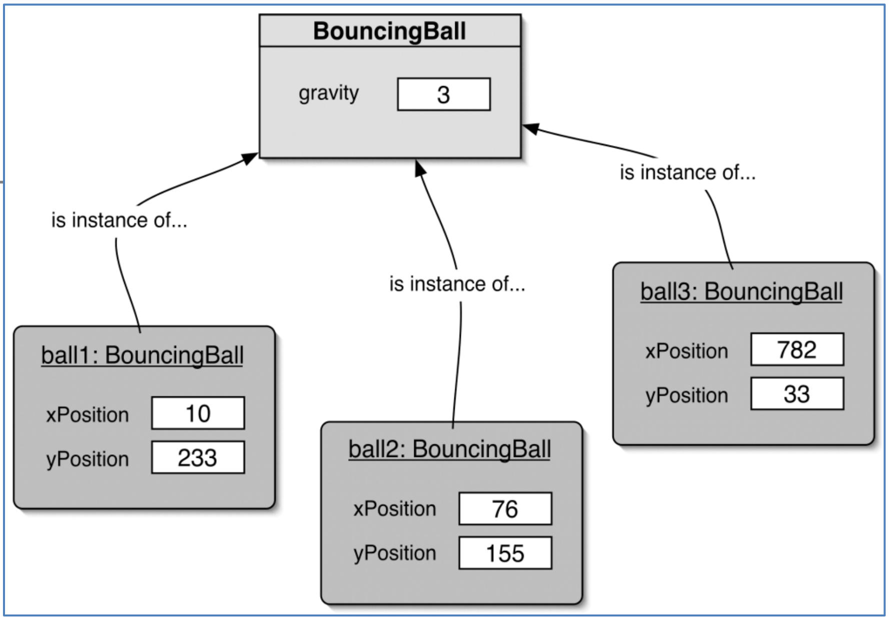
- Java supports static methods as well as static variables.

- Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

```
ClassName.methodName(args)
```

Static Methods

A common use for static methods is to access static fields.



For example, we could add a static method to the BouncingBall class to access the gravity static field:

```
public static int getGravity()
{
    return gravity;
}
```

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Writing class documentation

- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- Make your class a 'library class'!

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors’ names
- documentation for each constructor and each method

Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

javadoc

- The comment start symbol must be of this format in order to be recognised as a javadoc comment: /**
- Such a comment immediately preceding the:
  - class declaration is read as a class comment.
  - method signature is read as a method comment.
- Other special key symbols for formatting documentation include:

  @version
  @author
  @param
  @return

Javadoc – class comment

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (30.Mar.2006)
 */
public class Responder
{
```

Javadoc – method comment

```java
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return A set of Strings, where each String is
 *         one of the words typed by the user
 */
public HashSet<String> getInput(String prompt)
{
   ...
}
```

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

The danger lurking within!

Calculated data

```
public class Employee
{
    private double salary;
    private double deductions;
    private double netSalary;

    //code omitted

    public void calculateNetSalary()
    {
        netSalary = salary – deductions;
    }
    public void setSalary(double salary)
    {
        this.salary = salary;
    }
}
```

netSalary is calculated data.
→ what happens when we call the setSalary mutator? Is the netSalary field updated?

DATA INTEGRITY WARNING:

- netSalary can contain stale data.
- There is no need to have a netSalary variable; it can always call the netSalary method to get this value.
- We need to re-write calculateNetSalary() to reflect this.

Calculated data

```
public class Employee
{
    private double salary;
    private double deductions;
    :
    public double calculateNetSalary()
    {
        return (salary – deductions);
    }
    public void setSalary(double salary)
    {
        this.salary = salary;
    }
}
```

netSalary is no longer declared.

calculateNetSalary () now returns the result of the calculation.

→ No calculated data is stored, so no stale data!

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Debugger

- Errors in programs are called bugs. A debugger can be used to fix bugs; hence the name debugger!
- Most IDEs come with a debugger; BlueJ has one.

- *Demo of the debugger in BlueJ*

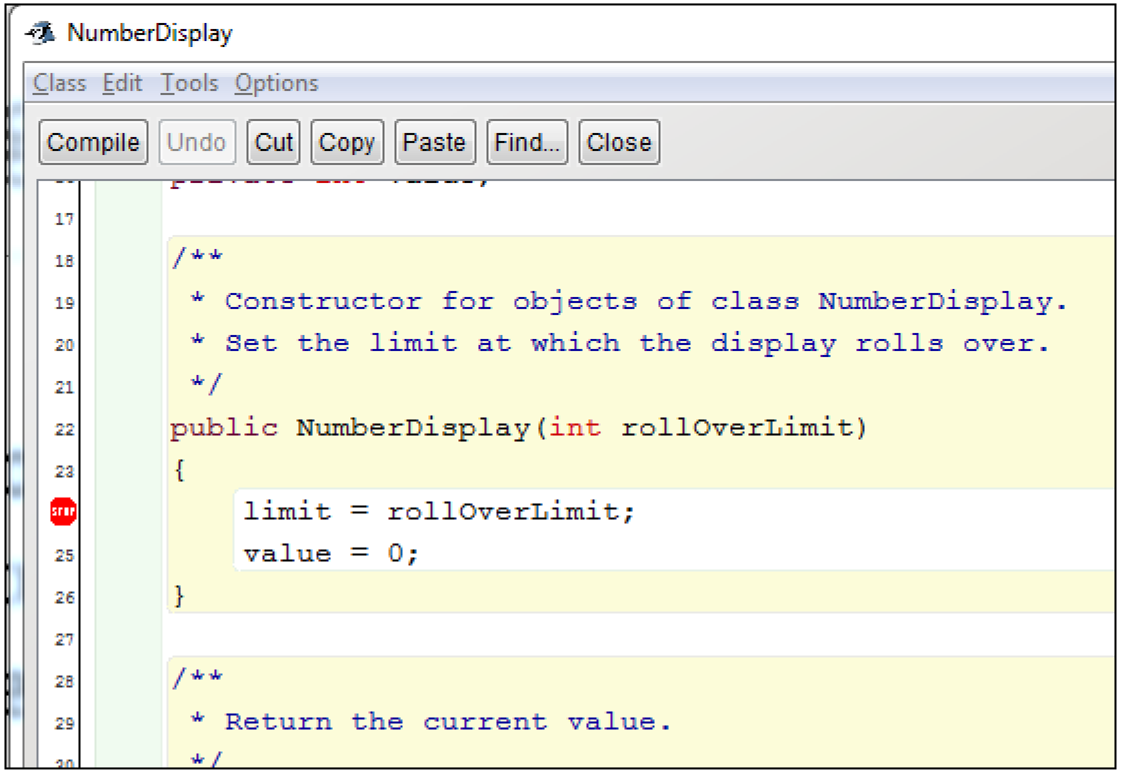
Debugger

- A debugger is a software tool that helps in examining how an application executes.
- It lets programmers execute an application one statement at a time.
- It typically provides functions to stop and start a program at selected points in the source code, and to examine the values of variables.

Debugger - instructions

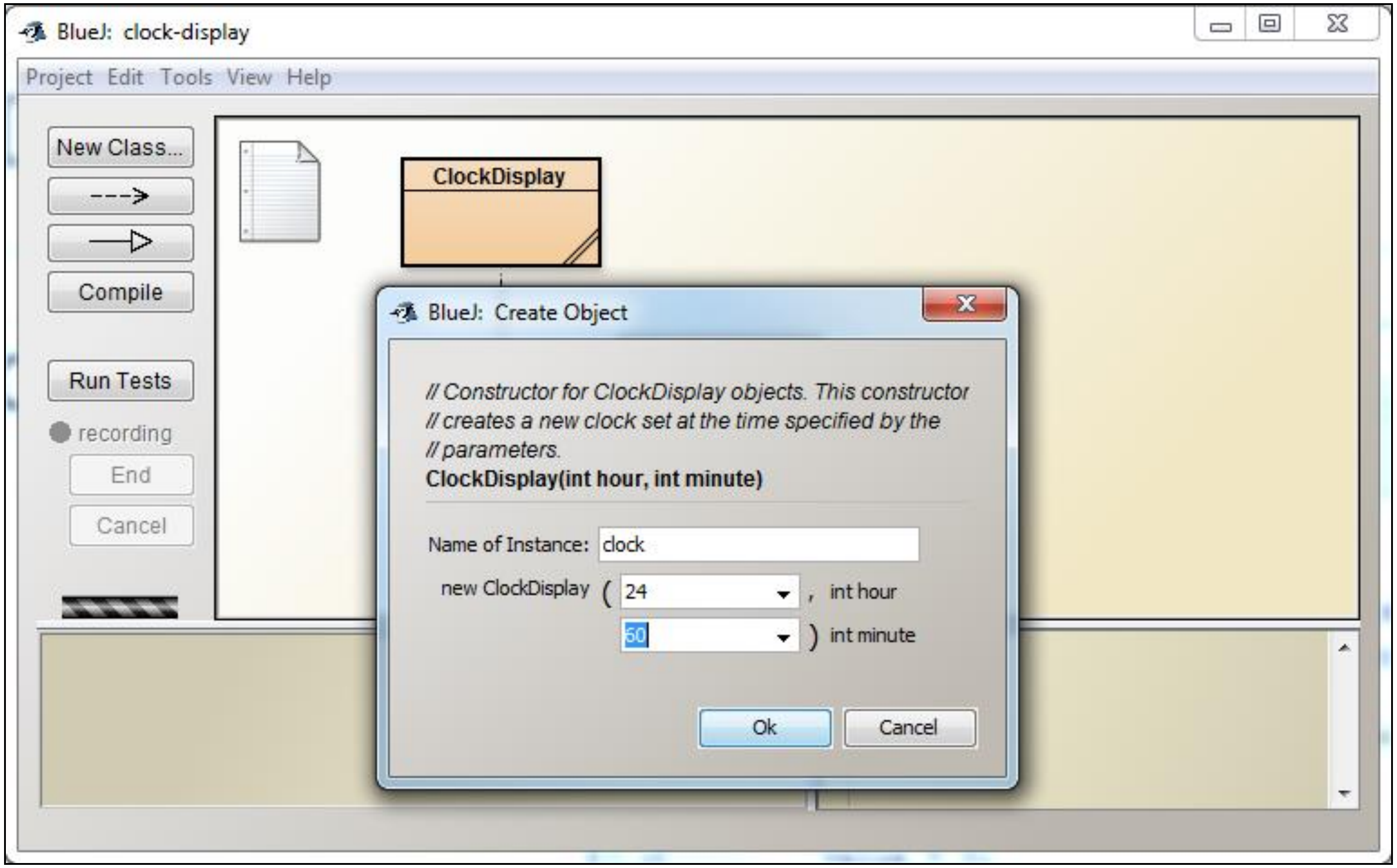
- Set a breakpoint on an executable statement (by clicking on the line number).

- To stop our program at this particular breakpoint, we need to create an instance of the **NumberDisplay** object:
  - As the ClockDisplay class creates two instances of NumberDisplay, we will create an instance of ClockDisplay.



```

    /**
     * Constructor for objects of class NumberDisplay.
     * Set the limit at which the display rolls over.
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
```

Debugger - instructions

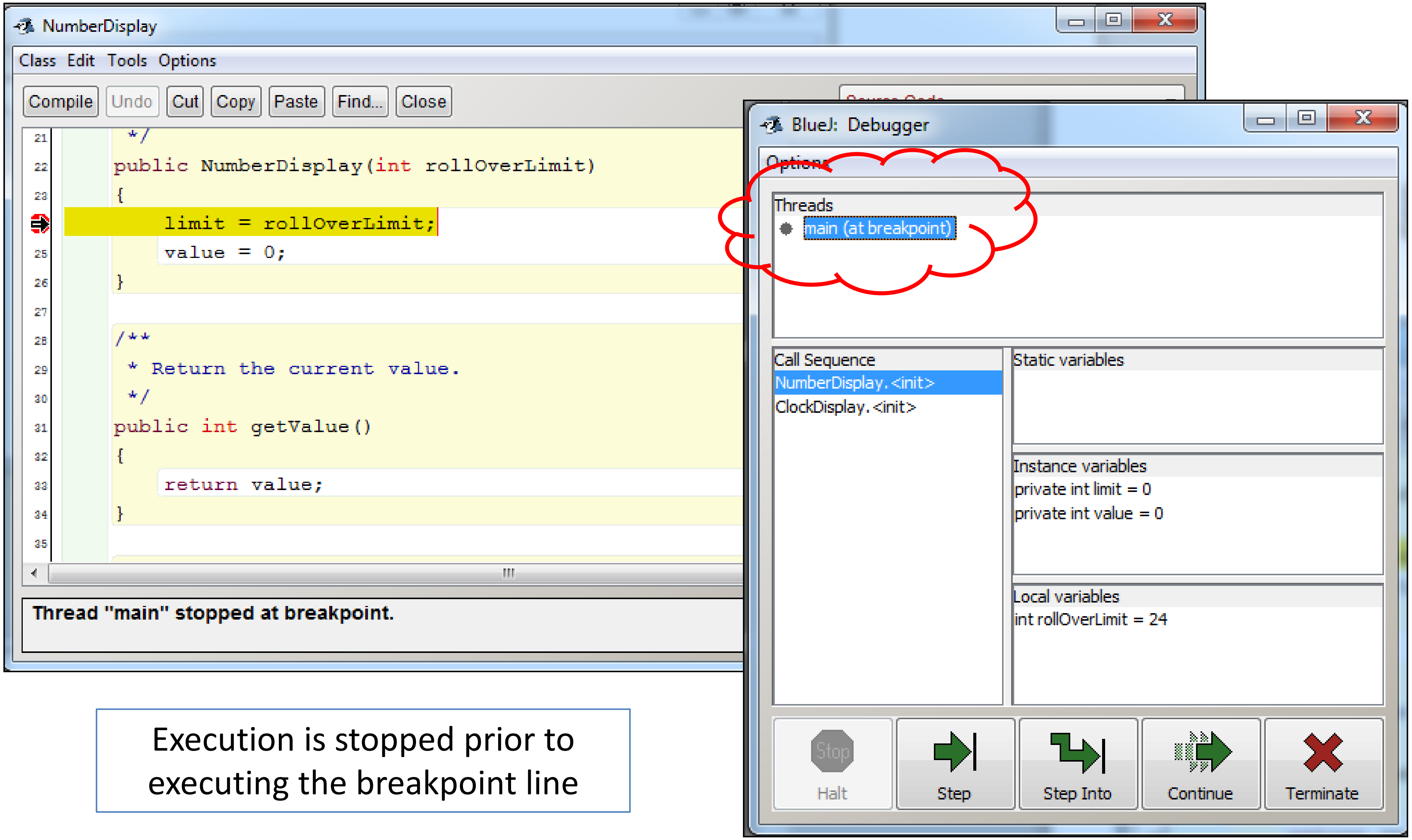


// Constructor for ClockDisplay objects. This constructor
// creates a new clock set at the time specified by the
// parameters.
ClockDisplay(int hour, int minute)

Name of Instance: clock

new ClockDisplay (24 , int hour
60) int minute

Debugger - instructions



```
    */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }
```

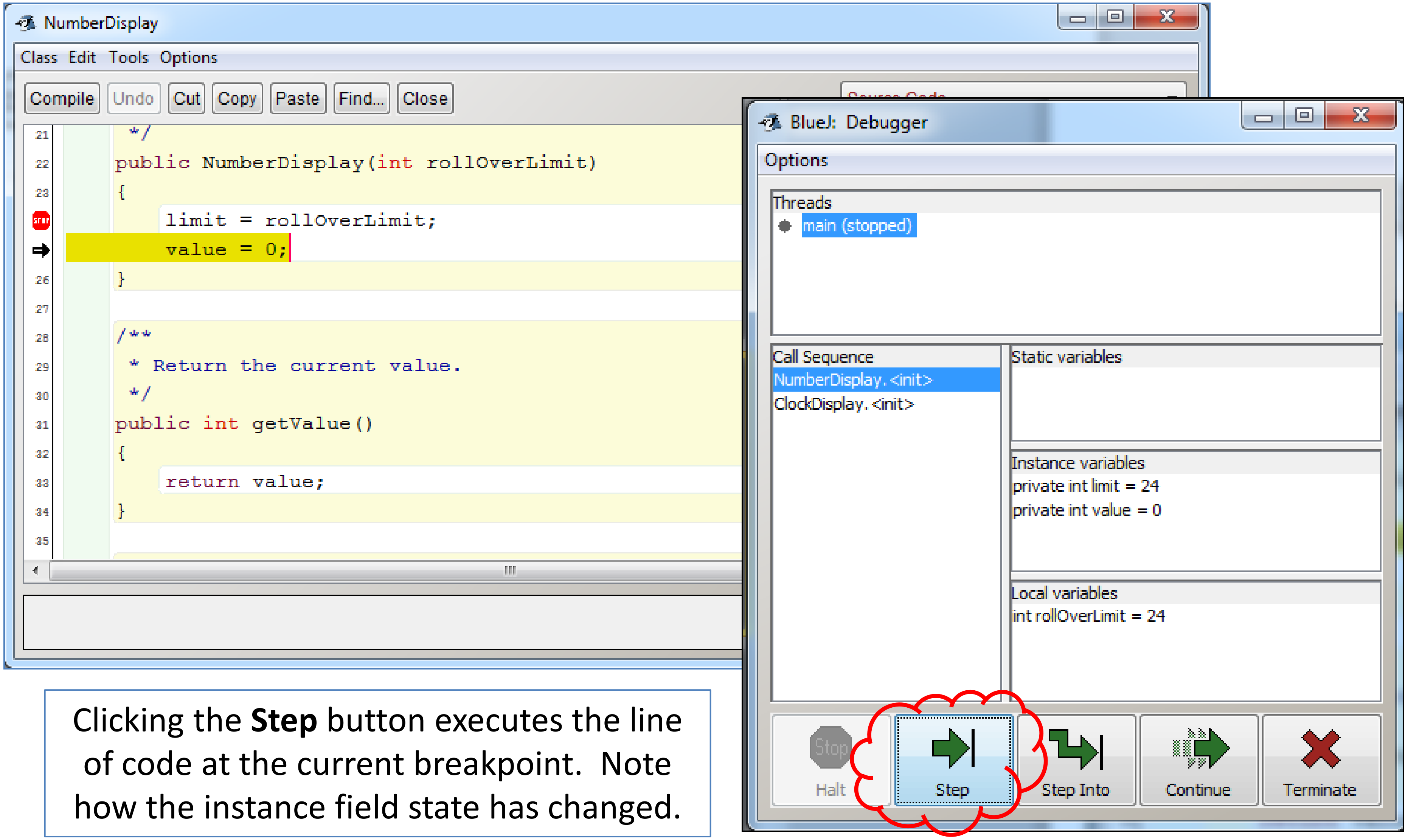
Thread "main" stopped at breakpoint.

Instance variables:
private int limit = 0
private int value = 0

Local variables:
int rollOverLimit = 24

Execution is stopped prior to executing the breakpoint line

Debugger - instructions



```
    */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }
```

Threads
main (stopped)

Call Sequence
NumberDisplay.<init>
ClockDisplay.<init>

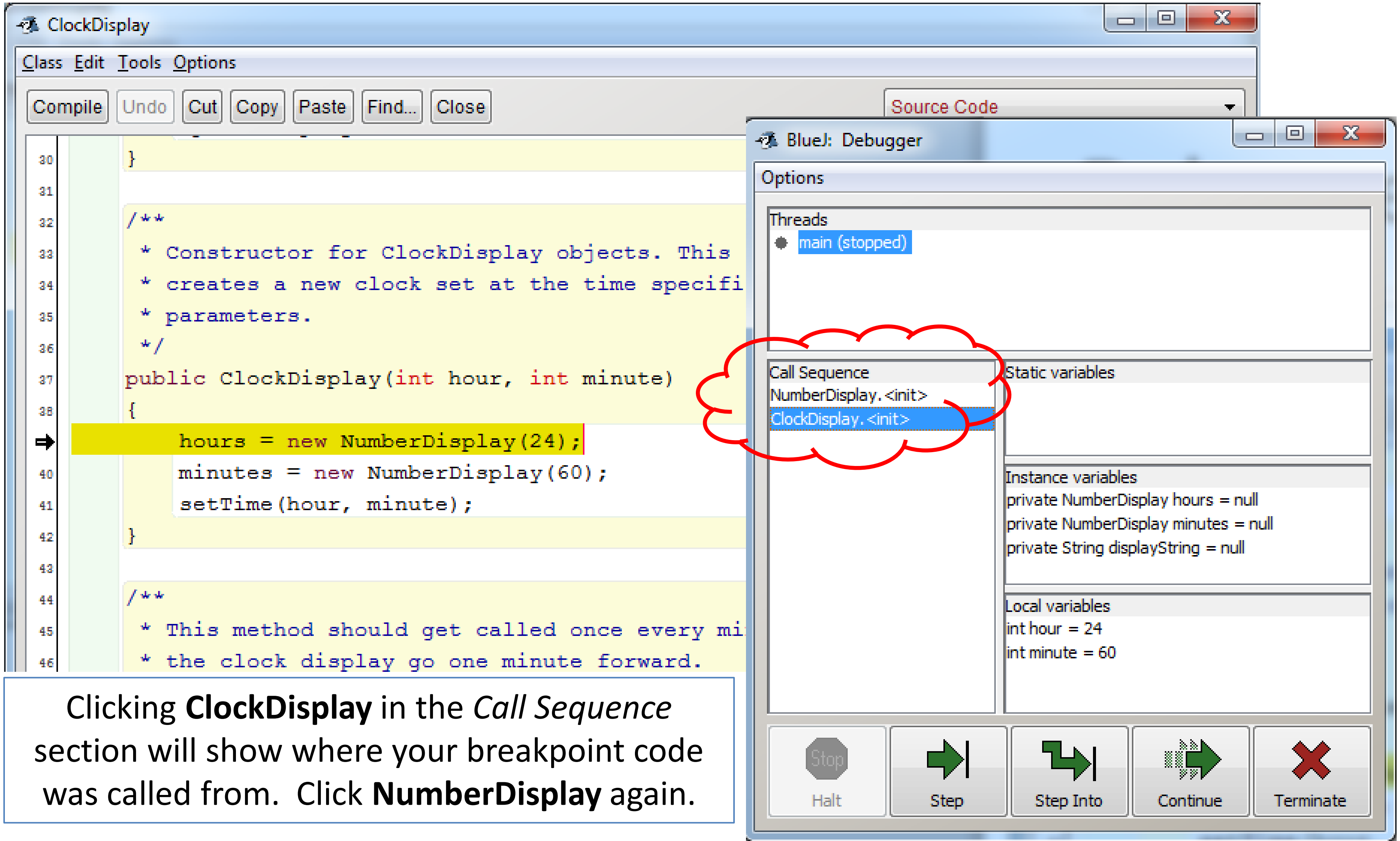
Static variables

Instance variables
private int limit = 24
private int value = 0

Local variables
int rollOverLimit = 24

Clicking the **Step** button executes the line of code at the current breakpoint. Note how the instance field state has changed.

Debugger - instructions



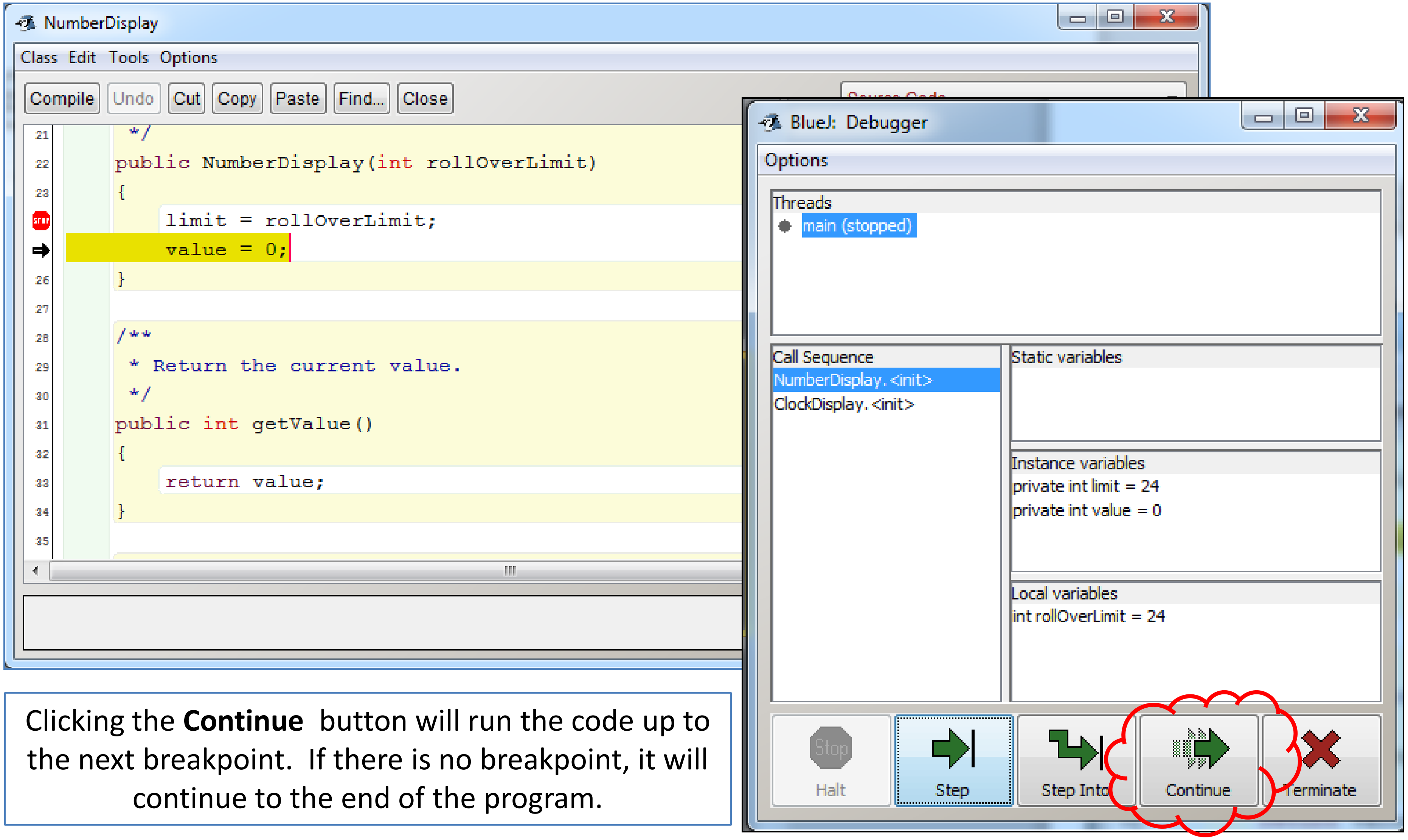
```
    }

    /**
     * Constructor for ClockDisplay objects. This
     * creates a new clock set at the time specifi
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every mi
     * the clock display go one minute forward.
```

Instance variables
private NumberDisplay hours = null
private NumberDisplay minutes = null
private String displayString = null

Local variables
int hour = 24
int minute = 60

Clicking **ClockDisplay** in the *Call Sequence* section will show where your breakpoint code was called from. Click **NumberDisplay** again.

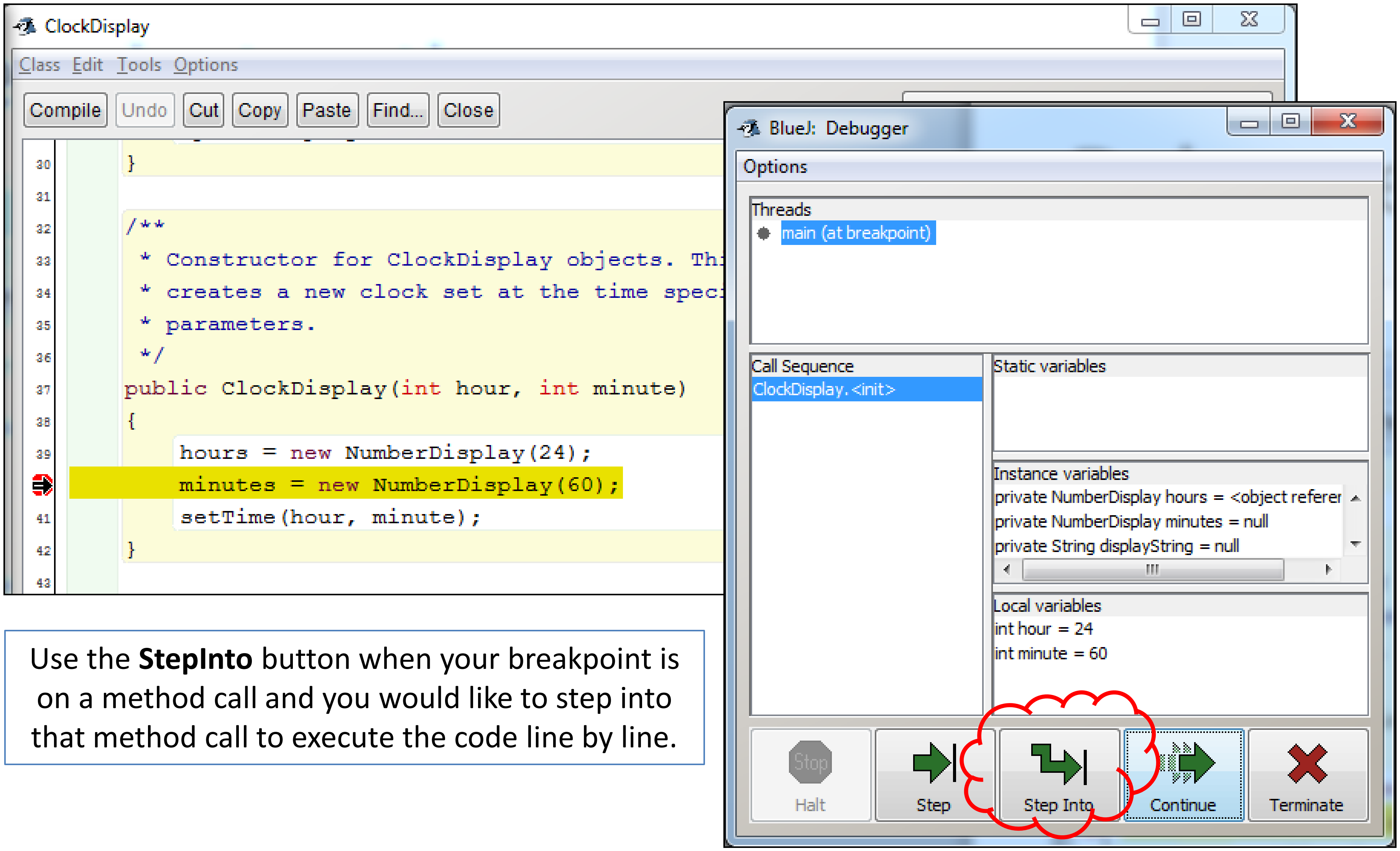
Debugger - instructions



```
    */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }
```

Clicking the **Continue** button will run the code up to the next breakpoint. If there is no breakpoint, it will continue to the end of the program.

Debugger - instructions



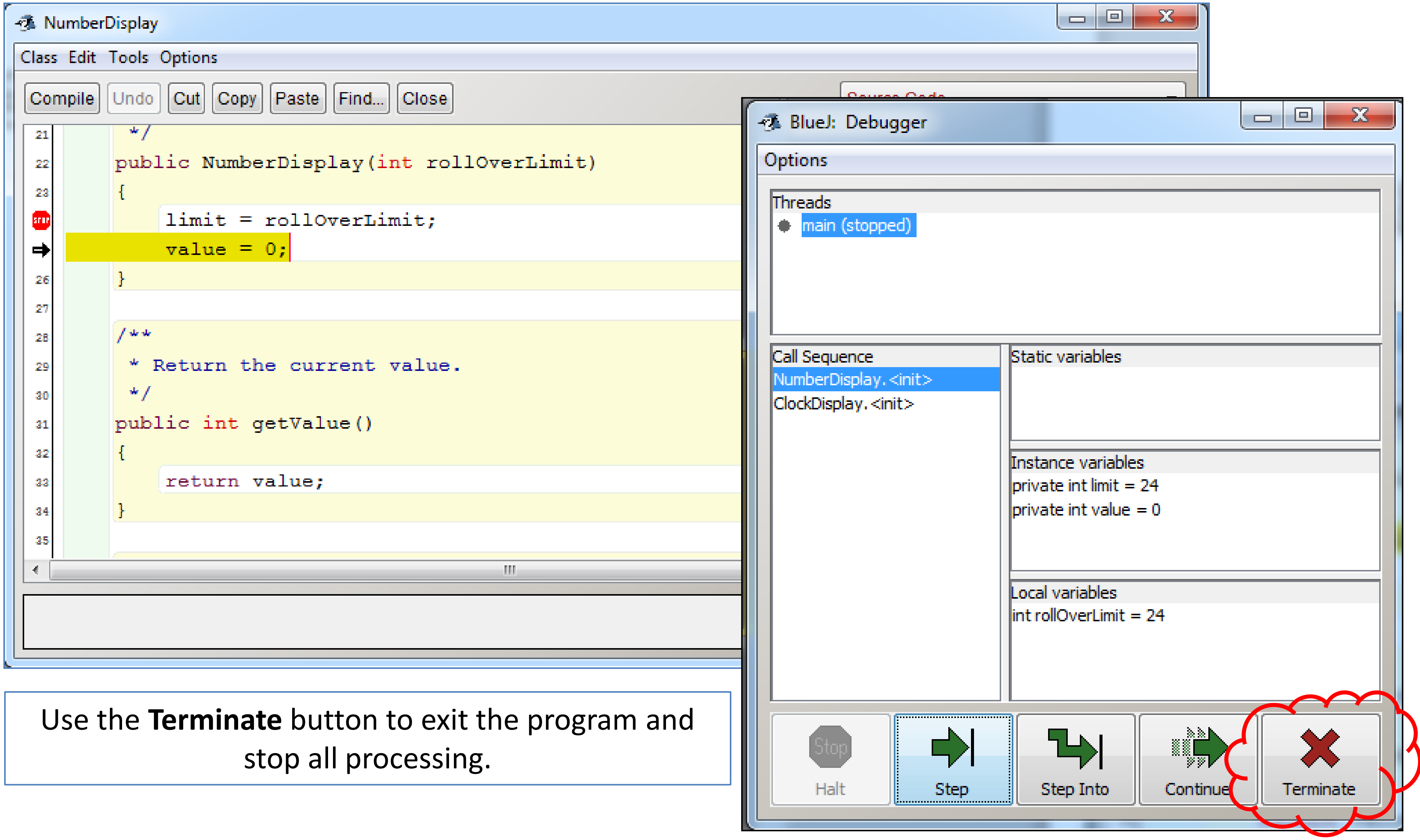
```
    }

    /**
     * Constructor for ClockDisplay objects. Thi
     * creates a new clock set at the time speci
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }
```

```
private NumberDisplay hours = <object referer
private NumberDisplay minutes = null
private String displayString = null
```

```
int hour = 24
int minute = 60
```

Use the **StepInto** button when your breakpoint is on a method call and you would like to step into that method call to execute the code line by line.

Debugger - instructions



```
    */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }
```

Threads
main (stopped)

Call Sequence
NumberDisplay.<init>
ClockDisplay.<init>

Static variables

Instance variables
private int limit = 24
private int value = 0

Local variables
int rollOverLimit = 24

Use the **Terminate** button to exit the program and stop all processing.

Debugger

- Debuggers are especially useful when your program contains logical errors.
  - Logical errors are errors that the compiler will not pickup but that lead to incorrect results e.g. if your syntax is correct but the logic of your problem solution is faulty.
- Using the debugger, you can trace how each of the calculations and changes made to fields/variables happen and hopefully figure where the error is occurring.

Topic List

- Typical Compilation Errors
- Multiple Constructors
- Static Variables
- Static Methods
- Javadoc (annotations)
- Storing calculated data
- Debugger
- Compound Assignment Statements

Compound assignment statements

```
balance += amount;
        is shorthand for
                balance = balance + amount;

balance -= amount;
        is shorthand for
                balance = balance - amount;
```

Compound assignment statements

	Full statement	Shortcut
Mathematical shortcuts	x = x + a;	x += a;
	x = x - a;	x -= a;
	x = x * a;	x *= a;
	x = x/a;	x /=a;
Increment shortcut	x = x+1;	x++;
Decrement shortcut	x = x - 1;	x--;

Questions?