Lecture Week 6

- GUIs, Swing, AWT
- Event Driven Programming
- Simple Swing/AWT GUI-Based Programs
- Broadcasts and Listeners – Alternate Syntaxes
- Multithreading
- More

GUIs, Swing, AWT, Event Driven Programming

Swing/AWT Why?

- Android
  - Does not use the Swing or AWT packages
    - It's GUI system is unique to the Android API although many Swing/AWT GUI concepts (not code) are transferrable
  - So why are we discussing them?
    - Before the landslide of new Android concepts we can introduce some important topics in a relatively simple code environment
    - In addition to common GUI concepts the concepts of event driven programming, broadcasting and listening can be introduced using Swing/AWT
    - The tricky syntax for listening and responding to events is the same in Android
    - Threading and its relationship to GUI responsiveness (very important in Android) can also be demonstrated

GUI, Swing , AWT

- Swing + AWT (Abstract Window Toolkit)
  - 2 Packages that contain classes + … that abstract the essentials of a GUI environment to allow applications programmers to create and use such an environment
  - AWT was the original GUI package
    - It's heavyweight – it's a thin interface to the OS's GUI code
      - All controls (buttons, drop down lists etc) are the OS's
  - Swing is lightweight
    - Uses AWT to create an OS window (heavyweight) then does all the GUI drawing inside this window itself (lightweight)
  - Swing classes replace many AWT classes but not all
    - Certain raw graphics classes (e.g. java.awt.Color) are still used
- GUI (Graphical User Interface)
  - Event-driven, Windows-based interfaces that handle user I/O

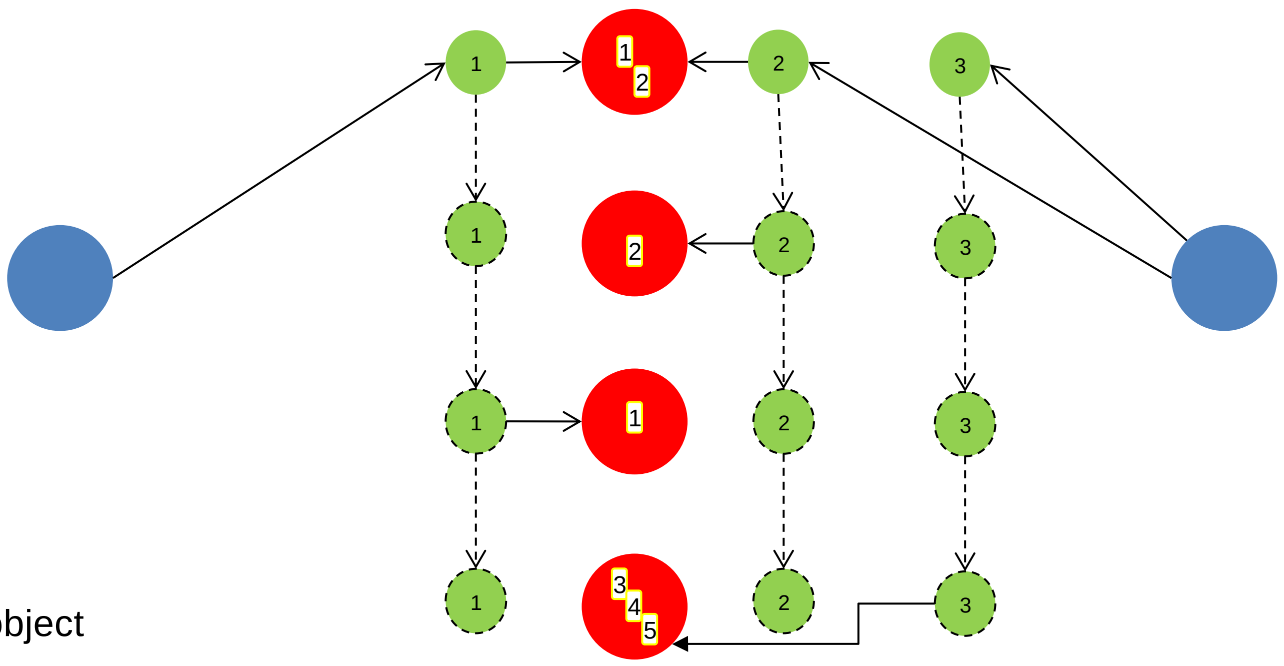
Events, Listeners

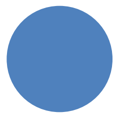
Examples:

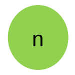
- Mouse down event, event object contains mouse down coordinates, event handler in a canvas object's class draws a dot at passed coordinates.
- System throws NullPointerException object, object contains exception data, hierarchy of catch blocks listen and potentially act (non-standard example, Why?)

- Event
  - An object (event object) broadcast by one object to all other objects (it is an abstraction of an actual event), the other objects can choose to listen or not
  - Sending object signals some particular event has happened
    - By sending a particular type of event object packed with data about this particular instance of the event type
  - Receiving objects (called listeners) responds by executing one of their methods (called an event handler) designated to execute in response to the reception of that particular type of event object
    - It will usually use the incoming event object's data
- A Listener
  - Can have multiple event handler methods that will each execute in response only to the reception of a particular type of event object
- Multiple Listeners
  - Can listen for the reception of the same event object in which case multiple event handler methods will execute in response to its reception at these multiple listeners (one per listener)

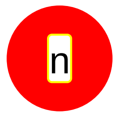
Broadcasters, Events, Listeners, Event Handlers



 Broadcasting object



Event object



Listening object
and
Event handling method

Event Driven Programming

- So Far
  - Programs control sequence of statement execution
    - Not entirely true as user console input could influence execution sequence
- GUI Code is Inherently Event Driven
  - We write code in event handling methods that will only execute as the result of the reception of a user (or system) event object
  - So at the method level we cannot control which method will execute next
    - The code in these methods better not have any sequencing dependencies
    - Of course our programming control structures still give us control over statement execution sequence within a method

Event Driven Programming

*All signalling classes have such methods otherwise what's the point of their existence

- Event Handler Methods
  - Are never called (invoked) from the code we write
    - Well rarely anyway
- How then are these Event Handler Methods Ever Called?
  - Signalling objects (e.g. buttons) require us to supply the address (reference variable) of a listener object of the appropriate type (class or interface) as a parameter to its listener-setting method*
    - This class/interface contains methods that can deal with each of the events the signalling object recognises
  - We supply an object of the appropriate type by extending/implementing the required class/interface and overriding/implementing inherited event handler methods coded with our custom responses
  - When the signaller detects an event it recognises, code internal to it (API code so far) invokes the appropriate event handler method on our supplied object causing our custom response code to execute
    - HOW? Code in the signalling class knows the address (reference variable) of our listener object (we supplied this) plus the names of methods we must have overridden or implemented (supplied object had the appropriate type)
    - Therefore it can invoke the appropriate method on the listener object
    - Therefore our custom code executes

Hooking Our Code into the API Code – HOW???

- What do you (or API code) need to know to invoke a method on an object
- 1. Its reference
  - This is just an address
- 2. Its type
  - So you can be sure the method invocation is syntactically (and therefore semantically) valid
    - No point in invoking a cook method on a refrigerator object

Hooking Our Code into the API Code – HOW???

- Consider
  - **`endButton.addActionListener(clickListener);`**
  - clickListener is an object reference
    - So API code (JButton class/addActionListener public method) now has a reference to an object that we instantiated in our code
  - What type of object?
    - addActionListener requires a parameter of type ActionListener
    - It's an interface type, the interface has one required method
      - It's actionPerformed(…)
    - clickListener therefore must be instantiated from a class that implements this interface
      - i.e. that contains an implemented actionPerformed(…) method
  - API code has a reference to an object it knows can have actionPerformed(…) on it
  - It can therefore call this method as required
    - i.e. when endButton is clicked
  - We coded the actionPerformed(…) method in a class that implemented the ActionListener interface
    - It's code is our response to endButton being clicked
    - Our response is therefore hooked into (has become part of) the API code!!!

First Swing Program

2 Classes:
FirstSwingDemo: Combined Driver and JFrame set up
EndButtonClickListener: Listener

- GO DO: FirstSwingDemo class
  - Code discussion
    - JFrame is heavyweight
      - i.e. it's an OS window (whatever OS you are on) that does all the things an OS window does (close minimise, move, resize etc …)
        - » We have blocked the normal close window button behaviour in our code. WHY?
      - Everything in the JFrame is lightweight
    - Import statements
      - e.g. swing, AWT as required
      - Swing is in "javax" NOT "java" package
    - JFrame class has methods (e.g. setDefaultCloseOperation) and also public constants that can be used as parameters to some of these methods (e.g. DO_NOTHING_ON_CLOSE)
      - It's all in the API reference

The default action is HIDE_ON_CLOSE. If we used the default and closed the window using its close button NOT the JButton what would happen?

First Swing Program

addActionListener(clickListener) – this is where we pass a reference to our listener object to swing/AWT. It knows the listener object class must implement actionPerformed(…) therefore when swing/AWT detects a button click it can invoke this method on our listener. Brilliant!!!

- GO DO: FirstSwingDemo class
  - Code discussion (cont.)
    - Adding swing controls (e.g. a JButton) to the JFrame
      - i.e. building a functional application window
    - A newly instantiated JFrame is initially invisible
    - The JButton constructor we use takes the button's text as its only parameter
    - The addActionListener method of the JButton class allows us to connect a listening object to the button
      - It's called registering the listener
      - It's argument is of the interface type java.awt.event.ActionListener
        - » This interface insists on one method to be implemented void actionPerformed(ActionEvent e)
        - » This method will be called (by Swing/AWT) when the button is clicked
        - » Swing/AWT supplies the event object (e) fully loaded with event data (we don't use any of it in this case)

First Swing Program

2 Classes:
FirstSwingDemo: Combined Driver and JFrame set up
EndButtonClickListener: Listener

- GO DO: FirstSwingDemo class
  - Code discussion (cont.)
    - Once you create a JFrame its lifetime extends until you end the program that created it (using System.exit(…) in this case)
    - As a consequence closing the JFrame does not terminate the application it's part of because the JFrame still exists in memory!!!
      - Unless: firstWindow.setDefaultCloseOperation(Jframe.EXIT_ON_CLOSE)
      - Why? In a GUI some new event may open the JFrame again
- GO DO: EndButtonClickListener class
  - It’s this class that is used to instantiate a listener object of type java.awt.event.ActionListener a reference to which is supplied as a parameter to the addActionListener method of the JButton
  - The type of the parameter is an interface type therefore the EndButtonClickListener class must implement the interface
  - The interface has one method (actionPerformed(…)) which will execute when the JButton is clicked
  - In it we simply end the program (System.exit(0)) which will end the GUI (recall we blocked ending the program by closing the JFrame related window)

First Swing Program Take 2

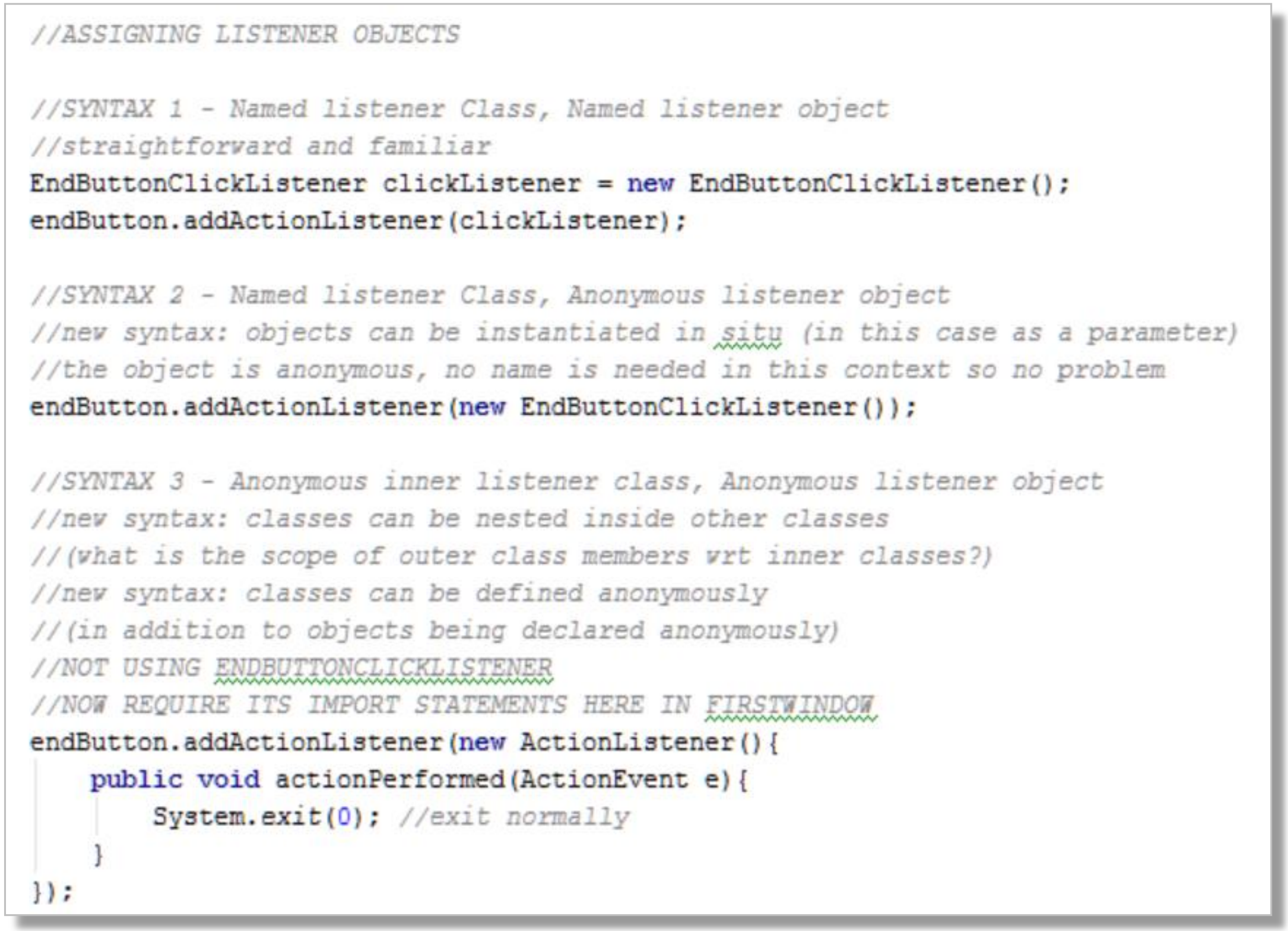
3 Classes:
FirstWindow: pure JFrame extender
FirstWindowDriver: pure Driver
EndButtonClickListener: Listener

- GO DO: FirstWindowRewrite + …
  - Code Discussion
    - This is the recommended way to create a new window
      - Define a class that extends JFrame then customise this class as required for your particular window
      - FirstWindow — Do all window initialisation and customisation in this class's constructor
      - EndButton-ClickEventListener — Code additional listener classes for any listener references assigned in the JFrame extender class
      - FirstWindowDriver — Instantiate your custom window and make it visible in code in a driver class (usually driver code knows when to make a window visible, the JFrame subclass code cannot presume to know this)
    - Notes:
      - Call to JFrame constructor (super) in custom window class's constructor
      - Inherited JFrame Methods are invoked on "this" which is assumed and so doesn't appear (e.g. setSize(…), add(…), …)
        - » this = the object that caused this code to run
      - Adding a title to a window
      - Various syntaxes for instantiating and assigning listener objects
        - » GO DO: FirstWindowRewriteSyntax + …

Various Syntaxes - Listener Objects

- Named Listener Class, Named Listener Object
  - Easy, EndButtonClickListener.java required
- Named Listener Class, Anonymous Listener Object
  - In situ instantiation of an anonymous object as a parameter
  - EndButtonClickListener.java required
- Anonymous Listener Class, Anonymous Listener Object
  - Difficult syntax BUT used all the time in Android
  - An anonymous inner class is defined and an anonymous object of that class is auto-instantiated in situ as a parameter
  - EndButtonClickListener.java NOT required

Various Syntaxes - Listener Objects



```
//ASSIGNING LISTENER OBJECTS

//SYNTAX 1 - Named listener Class, Named listener object
//straightforward and familiar
EndButtonClickListener clickListener = new EndButtonClickListener();
endButton.addActionListener(clickListener);

//SYNTAX 2 - Named listener Class, Anonymous listener object
//new syntax: objects can be instantiated in situ (in this case as a parameter)
//the object is anonymous, no name is needed in this context so no problem
endButton.addActionListener(new EndButtonClickListener());

//SYNTAX 3 - Anonymous inner listener class, Anonymous listener object
//new syntax: classes can be nested inside other classes
//(what is the scope of outer class members wrt inner classes?)
//new syntax: classes can be defined anonymously
//(in addition to objects being declared anonymously)
//NOT USING ENDBUTTONCLICKLISTENER
//NOW REQUIRE ITS IMPORT STATEMENTS HERE IN FIRSTWINDOW
endButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        System.exit(0); //exit normally
    }
});
```

Syntax 3 – Sheeesh!!!

new xxx(constructorParameters){…}
xxx: an interface to implement or a class to extend
If it's an interface then Object is the class extended

```
endButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            System.exit(0); //exit normally
        }
    });
```

This is a single statement terminated with a semi colon
It's a method invocation on an object (endButton) with no return value

```
endButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            System.exit(0); //exit normally
        }
    });
```

This is the only parameter of the method invocation. It's an in situ instantiation of an anonymous object of interface type ActionListener

This is an anonymous class definition.
It's a subclass of Object that implements the ActionListener interface. An instance of this class is auto instantiated as a result of the new operator

```
endButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            System.exit(0); //exit normally
        }
    });
```

```
endButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            System.exit(0); //exit normally
        }
    });
```

This is the only method required by the ActionListener Interface and thus of any class used to make objects of the ActionListener type

More Swing/AWT Fun

- GO DO: LabelAndColour
- Code discussion
  - Window’s close button now closes window
    - Why?
    - Closing one window closes both. Why?
  - Syntax for calling one constructor from another
  - JFrames have a ContentPane
    - The ContentPane object abstracts the JFrame’s interior
    - It’s peculiar to JFrames, in general JControls do not have one
  - Colours are objects of the Color class
    - The <u>C</u>olor class is in the package java.awt
      - » NOT the package java.awt.<u>c</u>olor
    - The <u>C</u>olor class supplies many public static final Color constants preset to a set of common colours

Layout Managers

- Interesting But NOT Important
  - When you add a control to a JFrame object you are actually adding an instance variable, that references that control, to the JFrame object
- Don’t Add Controls to JFrame Objects Directly
  - You have no control over their positioning
- Instead Add a LayoutManager object to the JFrame then Add Controls using the LayoutManager
  - Which manages the positioning of controls
  - Use an add method overload with 2 parameters

param1:
control to add
param2:
layout manager target

- There are Several Types of LayoutManager
  - e.g. BorderLayout (page start/end, line start/end, center)
    - Old area names are: north, south, east, west, center
  - e.g. FlowLayout (left to right, top to bottom)
  - e.g. GridLayout (like a table with rows, columns and cells)

GO DO: BorderLayoutManagerDriver

- GO DO: BorderLayoutManagerDriver + …
  - Code Discussion
    - The border layout manager's areas are quite fluid and are allocated space in the following order
      - Page start/end then line start/end then center
      - Unused spaces are devoured by regions later in the priority list
    - Adding a layout manager object to a JFrame
      - Anonymous, in-situ instantiation
    - There is always a bit of messing about in swing to get things to look right (e.g. the centering of the labels in their respective border layout regions)

JPanels

- JPanels
  - Controls can be added to a JPanel for grouping purposes
  - Panels can be added to Panels to create Panel hierarchies
  - Each JPanel can have its own Layout Manager
- GO DO: PanelDemo
  - Very bad code style
    - Everything has been wrapped into one class for teaching convenience
      - The driver class (see main)
      - The JFrame/Window class (see constructor)
        - » PanelDemo class extends JFrame class
      - The listener class (see actionPerformed method)
        - » PanelDemo class implements ActionListener interface
    - 3 different layout managers used
    - 1 listener deals with three messages from three different buttons. HOW?

Lots More Swing/AWT

- Images and Icons
  - GO DO: iconDemo package
- Menus, Scrollbars, Text Fields, Window Listeners
  - There are packages containing classes with examples of all of these to download (week 6 code)
- Drawing
  - GO DO: graphicsDemos Package
    - Custom drawing requires us to override the paint method which up till now has been called automatically to draw all of the swing components we have used whenever necessary
      - Actual drawing is done using the Graphics (actually the Graphics2D) object passed in as a parameter to paint
    - Since we are only scratching the surface of AWT/Swing these demo programs are not necessarily examples of best practice (we are not specialised AWT/Swing programmers)

Multithreading

- Unresponsive GUI
- Fixing It 1
- Fixing It 2

Multithreading

- How Come:
  - Your virus scanner can run while you are word processing?
  - Because they are running on different threads
- Thread
  - A separate computation process
- Are they running in Parallel
  - Possible if you have a multi-core processor in your machine
  - If not, a single core (usually under the direction of the OS) is executing multiple threads by allowing each to sequentially have a time slice of its resources
    - The chip is so fast that to humans its seems to be operating multiple threads in parallel NOT in short sequential bursts

GO DO FillDemo

- So Far:
  - When we execute a Java program it runs in a single default thread
- GO DO: FillDemo

Source: Absolute Java – W. Savitch - Pearson Education (ISBN0-321-33024-2)

  - Code Discussion
    - We interact with the default single thread for the first time
    - Thread.sleep(PAUSE)
      - Stops execution of the thread its issued on (in this case the default thread) for 100 milliseconds
      - Since its invoked on the Class we know it's a static method of the Thread class
      - Thread class is in java.lang package so no import statement required (hint that its fundamental!)
      - A sleeping thread can be interrupted by another thread in which case the sleep method throws an exception which is one of the exceptions Java insists is dealt with, hence the try catch block which avoids a compile time error

InterruptedExceptions do not play a part in FillDemo

GO DO FillDemo

– Code Discussion (cont.)
  - someComponent.getGraphics();
    - – Returns the Graphics object that someComponent receives as a parameter in its paint method
      - » This allows us, for the first time, to do painting in a component outside the component's paint method
  - What's the problem?
    - – Try closing the JFrame window, the GUI is unresponsive
    - – But the code says:
      - » setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    - – It's interesting to note that the part of the JFrame's behaviour that is the responsibility of the OS is fully responsive (move, resize etc.) but the part that is the responsibility of Java (close) is not

GO DO FillDemo

- Code Discussion (cont.)
  - What is happening?
  - Click Start button → actionPerformed(…) begins execution
    - GUI (Swing framework) now waits until actionPerformed(…) finishes before processing any queued events (such as a close button click)
  - fill() begins execution
    - But we have deliberately made it lengthy
    - actionPerformed is still executing having called fill() it's now waiting for fill() to finish
  - Finally fill() finishes, so now actionPerformed(…) can finish
  - Any events that have been queued while actionPerformed(…) was executing (such as a close button click) can finally be serviced by the Swing framework

Creating and Running a New Thread

- Creating
  - Instantiate an object of a custom class you have coded which extends the Thread class
  - Override the inherited run() method with the code you wish to execute in the new thread
- Running
  - myThreadInstance.start();
    - This performs some thread related admin then executes myThreadInstance's run() method
  - This thread runs "in parallel" with the thread from which it is started
    - In the current case from the main default thread

GO DO: ThreadedFillDemo

Source: Absolute Java – W. Savitch - Pearson Education (ISBN0-321-33024-2)

- GO DO: ThreadedFillDemo
  - Code Discussion
    - Instead of calling a method (fillDemo()) to execute the circle drawing code a thread (additional to the main default thread) is created that runs the same code
    - Now the main thread that processes GUI events and the new thread which draws circles share processing time
      - i.e. during the drawing of circles the main thread gets time to process a close window event

GO DO: ThreadedFillDemo2

- GO DO: ThreadedFillDemo2
  - Code Discussion
    - What if you want to create a thread but the code you want to execute requires you to extend another class other than Thread
      - Remember a Java class cannot inherit more than one parent class
    - Procedure
      - Implement the Runnable interface in the class you would usually inherit Thread into
        - » Doesn't matter if you are already implementing an interface since in Java a class can implement any number of interfaces (but extend at most 1 class)
      - Implement the interface's only method (run()) as promised
      - Now instantiate a Thread object but this time instead of the parameter-less constructor use the constructor which takes a Runnable object – it's this object's run method that will execute in the newly instantiated thread
      - Start the thread

More

- To Be Explained in Context
  - These will be covered in detail as they are encountered in use where they are more easily explained
- Collections
  - A Collection is an object that can contain or group other objects
    - e.g. an array
  - Java has a rich Collections framework
    - Framework = Interrelated Class and Interface hierarchies
    - e.g. ArrayLists (extensible arrays), Maps (groups key/value pairs)

More

- Generics
  - “In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.”
  - Benefits
    - Stronger type checking at compile time
    - Less down casting
      - e.g. after extraction from collections with ancestor types
    - Code reuse (by simply changing the selected generic type)
- Threads
  - Simulated or real parallel processing
  - Important in Android so the main GUI thread is not made unresponsive by other demanding processing tasks within an App
  - We have already covered the basics