

Graphical User Interfaces

Prof. dr. Paul De Bra
 Technische Universiteit Eindhoven
 Universiteit Antwerpen

Part 4: User Interface Development

- Principles of GUI hardware and software
- The Java Programming Language
 - Java basics, objects, exceptions, threads, synchronization
- Creating Graphical Interfaces with Java
 - Applets (AWT and Swing)
 - Internet aspects of applets
 - Handling events

Bitmap Display Hardware

- Raster graphics: screen = matrix of colored dots. Features of graphical display systems:
 - GDS can perform graphical operations;
 - Main CPU can access the display memory: *frame buffer*;
 - Contents of frame buffer is copied to screen 60 to 90 times per second (*refresh rate*).

Bitmap Display hardware: limitations

- lines look ragged, but angle is important



- letters look ragged, but *anti-aliasing* helps



Bitmap Display Hardware: color depth

- 1, 15, 16, 24, 32 bit: direct mapping to color
- 8 bit: use of *color map*: use of array with 256 cells of RGB colors; example:

| | |
|---|-----------------------|
| 0 | 00 00 00 (black) |
| 1 | FF FF FF (white) |
| 2 | FF 00 00 (bright red) |
| 3 | etc. |

Window Systems Software

- Acts like a *server*: it offers (GUI) *services*.
 - applications can request window creation:
 - position and size requested, but can be denied;
 - window system “hides” position from application.
 - some window systems offer *backing store*:
 - contents of window is remembered by system;
 - when an obscured window becomes exposed it can be redrawn by the window system.
 - without backing store the application must redraw.

Window Systems Software (cont.)

- Applications use library functions to:
 - paint pixels, lines, ovals, rectangles, etc.;
 - draw strings (in a given font and color);
 - set the shape of the cursor;
 - move, resize, raise and lower windows;
- The functions may:
 - perform the operations in the application;
 - give commands to the window system software to perform the operations.

Window Systems Software (cont.)

- Focus:
 - The window system determines which window or application has *keyboard focus*:
 - Keyboard input generates events for that application;
 - The application has to figure out which “input field” the input is for.
 - The window system determines which application receives *mouse events*:
 - The application can e.g. track mouse movement and ask to change the mouse pointer shape when the pointer is over one of its windows.

Window Toolkits / Widget Sets

- Higher level objects with “compound” behavior, e.g. button:
 - textual or graphical label;
 - pseudo 3D border through shading; (light=left top)
 - shading changes when pressed, to create a pseudo 3D look of a pressed button;
 - callback function called when pressed.
- Additional features, e.g. double buffering.
 - functions draw in off-screen buffer that is then copied to the screen: avoids “flickering”

Layered Architecture of GUIs

- 2.5D Layered architecture:
 - windows appear to lay on top of each other (through a pseudo 3D border) and can be raised and lowered;
 - buttons, menus, appear on top of the window (through a pseudo 3D border)
 - a window consists of a *frame*, a drawing *panel*, and on top of that come the buttons, menus, etc.

Layered Architecture of GUIs (cont.)

- Handling events:
 - The window system determines which application should receive an event;
 - When an application receives an event it sends it to the “topmost” widget (e.g. button, menu item)
 - When the widgets wishes to ignore the event it is sent “down” the stack of widgets until one is found that will handle the event.

The Java Programming Language

- Why Java in this course?
 - There are rich Java toolkits for building GUIs;
 - GUI can be functionally separated from “back-end” functionality;
 - GUI and “back-end” can run in separate threads so that the GUI is never dead during back-end activity;
 - Java is a simple, “familiar” object-oriented programming language;
 - Java is *robust*: there is a good mechanism for handling exceptions (and generating error messages).

The Java Programming Language (cont.)

- More benefits of using Java:
 - The Java runtime environment is *small*;
 - Java is platform independent and portable: machine “independent” bytecodes, and no machine or OS dependencies in source code;
 - Java is *distributed*: classes can be loaded over Internet;
 - Java programs are *secure*: a security environment is used to configure what a program is allowed and what not; (e.g. *sandbox* in browsers)
 - Java programs are *potentially fast*: suitable for optimization, and JIT compilers are available.

The Java Development Kit

- Download from www.javasoft.com:
 - contains *javac* Java compiler;
 - contains *jre* with *java* bytecode interpreter;
 - contains *rt.jar* standard “library”.
 - contains *appletviewer* for testing applets (without browser)
- We do *not* use a GUI Builder (ex. JBuilder):
 - much larger, requires high-end computer;
 - strongly suggests the use of non-standard widgets, layout managers, etc.
 - allows developer to “draw” the user-interface, as a horizontal prototype.

Standard data types

- **byte**: 8 bit two's compliment integer.
- **short**: 16 bit two's compliment integer.
- **int**: 32 bit two's compliment integer.
- **long**: 64 bit two's compliment integer.
- **float**: 32 bit floating point using IEEE 754-1985 standard.
- **double**: 64 bit floating point using IEEE 754-1985 standard.
- **char**: 16 bit Unicode character.
- **boolean**: two valued type, with values *true* and *false*.

Expressions (similar to C and C++)

- 3.141592654 is a (literal) expression of type **double**.
- (i>64) && (i<73) is an expression of type **boolean**.
- (k>128) ? 1 : 0 is an expression of type **integer**.
- i=i is an **integer** expression. (It is also a statement.)
- i=i, j=2 is only allowed as an expression in the initialization or continuation of a **for** loop.
- 'a', '\n' and '\u1234' are **character** expressions. They are also literals.
- object.method(arguments) is a method call.
- Different integral types can be mixed in an expression.
- Integral types, floats and doubles can be mixed in an expression.

Statements

- The **empty statement** does nothing.
- A **variable declaration** is a statement, with or without initialization.
- A **labeled statement** is a statement that begins with a label (a name followed by :)
- Most statements are **expressions statements**, like assignments, increments, decrements and method calls.
- A **selection statement** causes a choice to be made depending on a value. There are **if**, **if-else** and **switch-case-default** statements.

Control Flow

- **Blocks**: statements can be grouped using { and }; a block can appear everywhere a statement can; variable declarations can appear anywhere in a block;
- **if-else** identical to its C counterpart:
if (boolean-expr) statement1 else statement2
Careful: boolean expression, not integer.
- **while** and **do-while** are identical to their C counterparts;
- **for** is identical to its C counterpart:
for (init-expr; boolean-expr; incr-expr) statement
In init and incr the “,” operator is allowed. The three expressions are optional so **for (;) is allowed.**

Control Flow (cont.)

- **switch** is identical to the C construct.
- **Labels** are typically used on blocks and loops, for use with **break** and **continue**.
- **break**: used to exit from *any* block, not just a switch or loop. **break** can be followed by the label of the statement to break out of.
- **continue**: used to skip to the end of a loop's body and evaluate the boolean expression that controls the loop. A label can be used.
- **return**: terminates the execution of a method and returns to the invoker, possibly returning a value.

Control Flow: Example

```
int i, j;
outer:
for (i=0; i<100; ++i) {
    for (j=0; j<100; ++j) {
        // ...
        break; /* break out of inner loop */
        // ...
        break outer; /* break out of outer loop */
        // ...
        continue outer; /* continue outer loop */
        // ...
    }
    // ...
    break; /* break out of outer loop */
    // ...
}
}
```

Statements (cont.)

- **Iterations statements**: **for**, **while** and **do while**
 - A construct like **while (i) --i;** is legal in C but not in Java. (why not?)
- **Jump statements**: **break**, **continue**, **return**, **throw**, but no **goto**;
- A **synchronization statement** prevents simultaneous use of variables in multi-threaded applications.
- A **guarding statement**, like **try**, **catch** and **finally** is used to handle exceptions.

Classes

- Collection of objects with same structure and behavior:
 - data fields: separate for every object;
 - methods: perform operations on data fields and parameters, optionally return a value;
 - static data fields: shared between all objects of the same class;
 - static class methods: can only use static data fields;
 - to avoid confusion: **this** used to refer to current object

Classes: Example

```
class Point {
    public int x, y;
    /* default constructor */
    public Point() {
        x = 0; y = 0;
    }
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}

class Rectangle {
    public Point origin, corner;
    public int Width() {
        return corner.x - origin.x;
    }
    public int Height() {
        return corner.y - origin.y;
    }
    public int Circumference() {
        return 2 * (Width() + Height());
    }
    public int Surface() {
        return Width() * Height();
    }
}
```

Inheritance

- A class may extend **one** other class by adding methods and fields and by replacing (overriding) methods and fields by new implementations. The extended class is called a **subclass**. The original class is the **superclass**.
 - methods of the superclass can be used in the subclass as if they belong to the subclass;
 - the word **super** can be used to use overridden features of the superclass;
 - *single inheritance*.

Inheritance: Example

```
class Pixel extends Point {
    public Color color;
    public Pixel(x, y, col) {
        super(x, y); color = col;
    }
}
```

- Note the use of **super**.
- In the earlier example a rectangle can be created based on an origin and corner object of class Pixel.

Visibility of fields and methods

- **int size;**
 - field can be accessed within the same package;
- **public int size;**
 - field can be accessed from anywhere;
- **private int size;**
 - field can only be accessed within same class;
- **protected int size;**
 - field can only be accessed within same class and subclasses;

Visibility: Example

```
class Rectangle {
    private Point origin, corner;
    /* make sure origin is left top and corner of right bottom */
    public Rectangle(Point org, Point cor) {
        if (org.x <= cor.x)
            if (org.y <= cor.y) {
                origin = new Point(org.x, org.y); corner = new Point(cor.x, cor.y);
            } else {
                origin = new Point(org.x, cor.y); corner = new Point(cor.x, org.y);
            } else if (org.y <= cor.y) {
                origin = new Point(cor.x, org.y); corner = new Point(org.x, cor.y);
            } else {
                origin = new Point(cor.x, cor.y); corner = new Point(org.x, org.y);
            }
    }
}
```

Creating and Destroying Objects

- All classes extend the **Object** class:
 - Inherit methods **equals(Object obj)**, **hashCode()**, **clone()**, and **getClass()**.
 - Objects are created by calling the **constructor** with the **new** keyword. Example: **p = new Point(2, 3)**; The default constructor does nothing.
 - Only the first statement of the constructor may call a constructor of the superclass: Example: **super(2, 3)**;
 - Objects cannot be explicitly destroyed. The **finalize** method can perform some actions when an object is being destroyed by the garbage collector.

Creating and Destroying Objects: Example

```
public class MyFile {
    private Stream File;
    public MyFile(String path) {
        File = new Stream(path);
    }
    // ...
    public void close() {
        if (File != null) {
            File.close();
            File = null;
        }
    }
    protected void finalize() throws Throwable {
        super.finalize();
        close();
    }
}
```

Abstract Classes

- In an **abstract** class some features are not implemented. Subclasses need to implement these features.
- ```
abstract class AbstrRectangle {
 public Point origin;
 /* don't know whether the corner or the width and height will be stored */
 abstract Point Corner();
 abstract int Width();
 abstract int Height();
 public int Surface() {
 return Width() * Height();
 }
}
```

## Abstract Classes (cont.)

```
public class Rectangle extends AbstrRectangle {
 /* we choose to store the corner */
 private Point corner;
 public Point Corner() {
 return corner;
 }
 public int Width() {
 return corner.x - origin.x;
 }
 public int Height() {
 return corner.y - origin.y;
 }
}
```

## Interfaces (semi-multiple inheritance)

- A class can inherit from only one superclass, but may implement several interfaces.

```
interface Debugging {
 void setDebug(boolean on);
 boolean Debug();
}
class Rectangle implements Debugging {
 private boolean debugmode = false;
 void setDebug(boolean on) {
 debugmode = on;
 }
 boolean Debug() {
 return debugmode;
 }
 // ...
}
```

## Arrays

- Arrays are objects:
  - declared without size, initialized with size:
 

```
int[] ia = new int[3];
float[][] mat = new float[4][4];
```
  - **length** field used to know last valid index:
 

```
for (int y=0; y<mat.length; ++y) {
 for (int x=0; x<mat[y].length; ++x)
 // ...
}
```
  - initialization mostly like in C.

## Strings

- Strings are *not* arrays! There are two kinds:
  - **String**: read-only
  - **StringBuffer**: mutable strings
- **str[i]** is invalid, so is **str.length**
  - **str.length()** is a method returning the length of the string
  - **str.charAt(i)** returns the character at position i
- Strings contain 16 bit characters

## String Operations

- **indexOf(char ch)** returns the first position of **ch**.
- **indexOf(char ch, int start)** first position of **ch** after **start**.
- **indexOf(String s)** first position of **s**.
- **indexOf(String s, int start)** first position of **s** after **start**.
- **lastIndexOf(char ch)** the last position of **ch**.
- **equals(String s)** tests string equality.
- **equalsIgnoreCase(String s)** tests string equality, up to the difference between upper- and lowercase.
- **substring(int beginIndex, int endIndex)** generates a substring (i.e. a new String object).
- etc.

## StringBuffer Operations

- **append(...)** appends a string representation of a boolean, character, integer, float, etc. to a StringBuffer.
- **insert(int pos, ...)** inserts a string representation of a boolean, character, integer, float, etc. into a StringBuffer at position **pos**.
- **toString()** converts a StringBuffer to a String.
- It is often more efficient to create and manipulate a StringBuffer, rather than operating on Strings, because of the overhead of object creation.

## Input and Output

- Typical examples of I/O classes are:
  - **FileReader** and **FileWriter**: read from or write to a file. Example: `f = new FileReader("myinput");`
  - **BufferedReader** and **BufferedWriter** are extensions to **FilterReader** and **FilterWriter**. Another example of filtered I/O is the **LineNumberReader**.  
`LineNumberReader in = new LineNumberReader(f);`  
`// ...`  
`System.out.println(in.getLineNumber());`
- Piped streams are used as input/output pairs.
- A **StreamTokenizer** is used to perform very elementary parsing on input.

## Exceptions

- Allow to separate error handling from normal operations (avoids clutter). Exceptions are *thrown* and *caught*.
  - A new exception class by convention extends **Exception** (which extends **Throwable**).  
`public class MyException extends Exception {`  
`public String reason;`  
`/* constructor */`  
`MyException(String s) {`  
`super("MyException with reason: " + s + ".");`  
`reason = s;`  
`}`  
`}`

## Exceptions (cont.)

- A method must declare the exceptions it may throw (unless it catches these exceptions itself):  
`public void doit(int i) throws MyException {`  
`if (i == 0) throw MyException("argument was 0");`  
`}`
- Exceptions are caught by enclosing code in **try** blocks.  
`try {`  
`int i = 0; doit(i);`  
`System.out.println("There was no exception");`  
`} catch (MyException e) { // handle the exception`  
`System.out.println("An exception was generated");`  
`}`

## Exceptions (cont.)

- A **try** statement may end with a **finally** clause, which will be executed at the end, *whether an exception was thrown or not*. The **finally** clause will be executed even if the **try** clause tries to return from the method.  
`try {`  
`int i = 1; doit(i);`  
`return 1; // try to return 1, but won't work`  
`} catch (MyException e) { // do nothing for now`  
`} finally {`  
`return 2; // this will be returned in any case`  
`}`

## Threads

- A single program can have multiple subprocesses running in parallel through the class **Thread** or the interface **Runnable**. The class **Thread** provides:
  - **run()** contains the code that does the actual work.
  - **start()** activates the thread by calling the **run** method.
  - **stop()** kills the thread.
  - **suspend()** stops the execution of the thread.
  - **resume()** resumes the execution of a suspended thread.
  - **setPriority(int pri)** sets the priority of the thread. A busy wait should set its priority to **MIN\_PRIORITY**.
  - **sleep(long mills)** suspends the thread during a number of milliseconds.

## Threads: Example

```
class PingPong extends Thread {
 String word; // what to print
 int delay; // how long to pause
 /* constructor */
 PingPong(String word, int delay) {
 this.word = word;
 this.delay = delay;
 }
 public void run() {
 try {
 for (;;) {
 System.out.print(word + "
");
 sleep(delay);
 }
 } catch (InterruptedException e) {
 return; // end this thread
 }
 }
 public static void main(String[] args) {
 new PingPong("ping", 33).start();
 new PingPong("pong", 100).start();
 }
}
```

## Alternative Example with Runnable

```
class PingPong implements Runnable {
 String word; // what to print
 int delay; // how long to pause
 /* same constructor */
 /* same run() method but with Thread.sleep(delay),
 because
 we're not a Thread object */
 public static void main(String[] args) {
 Runnable ping = new PingPong("ping", 33);
 Runnable pong = new PingPong("pong", 100);
 new Thread(ping).start();
 new Thread(pong).start();
 }
}
```

## Synchronization

- Locking is used to avoid simultaneous access to objects or code by different threads:
  - A **synchronized** method cannot be simultaneously executed by two threads;
  - Exclusive access to an object can also be done in a block:
 

```
synchronized (balance) {
 balance += amount;
 }
```
- Careful: never synchronize on an object of class Thread!

## Synchronization (cont.)

- Threads may explicitly start each other:
  - **wait(long millis)** must always occur in a loop and will wait until notified or until the timeout (when it generates an **InterruptedException**). **wait(o)** and **wait()** mean there is no timeout.
  - **notify()** wakes up one of the waiting processes. (you don't know which!)
  - **notifyAll()** wakes up all the waiting processes.

## Synchronization: Example

```
synchronized void doWhenCondition() {
 while (!condition)
 wait();
 // do something useful after condition is true
}
synchronized void changeCondition() {
 // change a condition
 notify(); // or notifyAll()
}
```

## Java Applications and Applets

- Applets are embedded in an HTML page:
  - **Example** ("Rolodex" applet)
- Applications are stand-alone; they have a "main" method.
- One Java program can be used for both; this is explained on the [Java developer site](http://developer.java.sun.com/developer/technicalArticles/Programming/TurningAnApplet/).  
<http://developer.java.sun.com/developer/technicalArticles/Programming/TurningAnApplet/>

## Abstract Window Toolkit (AWT)

- It offers widgets and features that are available (and functionally but not visually identical) on all computing platforms.
- It uses the computing platform's window system for some of its operations.
- It contains some "native" code and is thus not 100% portable.
- A tutorial on writing AWT applets can be found at: <http://java.sun.com/docs/books/tutorial/applet/index.html>

## The Swing Toolkit

- It offers a *Pluggable Look & Feel*. User-Interfaces can have the same look and feel on all computing platforms.
- It does not use the computing platform's window system for most operations.
- It is implemented using 100% "pure" Java. There is no native code in Swing.
- Many Swing components are **not thread safe**.
- A Swing tutorial can be found at: <http://java.sun.com/docs/books/tutorial/uiswing/index.html>

## AWT vs. Swing: Common Properties

- They offer an event model that is closely tied to the object-oriented and layered model for user-interfaces.
- They offer classes for communicating with a Web-browser and with remote machines over Internet, for loading and manipulating images, and for loading and playing sound.
- They conform to the JavaBeans standard for reusable components.

## Combining AWT and Swing

- AWT uses *heavyweight* components:
  - AWT places components on top of an application's main window through the window system.
- Swing uses *lightweight* components:
  - Swing draws components on top of the application's main window by itself.
- AWT and Swing cannot be mixed. (Why?)

## The Hello World Applet

- This example draws a string directly in the applet's "space". We have an HTML file with an [alternative in AWT and Swing](#).

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorldApplet extends Applet {
 public void paint(Graphics g) {
 g.drawString("Hello world!", 50, 25);
 }
}
```

## Reading Parameters from HTML

- Information can be passed from the HTML file to the applet:
  - add a "param" tag to the HTML file:
 

```
<param name="string" value="Goodbye world!">
```
  - add a "getParameter" call to the Java code:
 

```
inputstring = getParameter("string");
```
- We have a HTML file with the [DrawStringApplet](#).

## Starting and Restarting Applets

- The **init** method is called:
  - the first time a page with the applet is loaded;
  - each time that page is reloaded from the server (parameters may have changed);
  - each time that page is revisited after the applet was destroyed (garbage collected).
- The **start** method is called:
  - after the first call to **init**;
  - each time the page is revisited (right after being resumed);
- The **paint** method is called:
  - when an AWT applet needs to be redrawn;

## Stopping and Destroying an Applet

- The **stop** method is called:
  - when the page containing the applet is “removed” (e.g. when the user follows a link);
  - after **stop** the applet’s main thread is suspended, but **stop** needs to suspend or stop additional threads the applet created;
- The **destroy** method is called:
  - when the Java virtual machine needs to reclaim memory and wants to garbage collect the applet.
  - it allows the applet to “do something” before being destroyed.

## Stop: Example

```
public void stop() {
 Socket s = null;
 try {
 s = new Socket(host, 80);
 PrintStream out = new PrintStream(s.getOutputStream());
 out.println("GET " + getParameter("STOPURL") + "
HTTP/1.0");
 out.println(""); out.println("");
 } catch (IOException e) { System.out.println(e); }
 finally {
 try { if (s != null) s.close(); }
 catch (IOException e2) { ; }
 }
}
```

## Drawing Things

- The AWT class **Graphics** defines drawing operations (for on- or off-screen images).
  - Because it also works on off-screen images it can also be used to draw things in Swing applets.
  - Each Swing component has a method **paintComponent(Graphics g)** in which you can place custom painting code.
  - All operations only have effect within the applet’s window (or off-screen image). They are automatically clipped.
  - The operations use properties from the graphic context of the component, like the color and font.

## Drawing Things (cont.)

- Some commonly used methods are:
  - **drawString(String s, int x, int y)**  
draws a string at a certain position.
  - **drawLine(int x1, int y1, int x2, int y2)**  
draws a line between two endpoints.
  - **drawRect(int x, int y, int width, int height)**  
draws an open rectangle with given upper lefthand corner, width and height.
  - **drawImage(Image img, int x, int y, ImageObserver obs)**  
draws an image with given upper lefthand corner. It interacts with an imageobserver for loading the image, because drawing can start before an image is loaded.

## Interaction with the Browser and the Web

- Through methods of the Applet class:
  - **public URL getDocumentBase()**  
returns the URL of the page containing the applet.
  - **public Image getImage(URL url, String name)**  
returns an **Image** object that can be painted on the screen. The name of the image is relative to the url.
  - **public AudioClip getAudioClip(URL url, String name)**  
returns an **AudioClip** object.
  - **public void play(URL url, String name)**  
loads and plays an audio clip.
  - **public String getParameter(String name)**
  - **public void resize(int width, int height)**

## More Interaction with the Browser

- Through the **AppletContext**:
  - obtained through **getAppletContext()**;
  - **public void showDocument(URL url)**  
requests that the browser show the indicated Web page instead of the page containing the applet.
  - **public void showDocument(URL url, String target)**  
requests that the browser show the indicated Web page. The target determines the frame to use. (“\_self”, “\_parent”, “\_top”, “\_blank”, named window).
  - **public Applet getApplet(String name)**  
returns the Applet with the given name.

## Communication Between Applets

- Through method calls:
  - only works with applets in the same HTML page (same frame), found through **getApplet**;
  - we have a [“client-server”](#) example.
- Through static data fields:
  - works across HTML pages *if* there is only one Java virtual machine in the browser.
  - we have a [“initialization”](#) example.

## A "simple" Example: the Rolodex

- The Rolodex applet performs the following functions:
  - images are scrolled "LEFT", "RIGHT", "UP" or "DOWN".
  - delay between frames ("nap") and between images ("pause") can be configured.
  - A soundtrack is played continuously or per image.
  - With each image a URL is associated. When the user clicks on the image the corresponding document is loaded. (Without URL clicking stops/restarts the display.)
  - The [source code of Rolodex](#) contains all the details.

## Rolodex Thread Management

- Applets cannot extend the **Thread** class but must implement the **Runnable** interface.
  - The applet declares a thread field:  
**Thread engine = null;**
  - The **init()** method (re)sets engine to null.
  - The **start()** and **stop()** methods of the applet must start and stop the animation thread.

```
public void start() {
 if (engine == null) {
 engine = new
 Thread(this);
 engine.start();
 }
}

public void stop() {
 if (engine != null && engine.isAlive())
 engine.stop();
 engine = null;
}
```

## Loading Images (AWT)

- A **MediaTracker** object is used to synchronize with the image being loaded:

```
Mediatracker tracker = new MediaTracker(this);
boolean fetchImages(Array imageurls) {
 for (int i=0; i<imageurls.length; ++i) {
 images[i] = getImage(imageurls[i]);
 // give tracker id of 0
 tracker.addImage(im, 0);
 } // now wait until all images loaded
 try {
 tracker.waitForID(0);
 } catch (InterruptedException e) {}
 return !tracker.isErrorID(0);
}
```

## Drawing Images on the Screen (AWT)

```
Image offScrImage = null; Graphics offScrGC = null;
public void paint(Graphics g) {
 // ...
 offScrImage = createImage(appWidth, appHeight);
 offScrGC = offScrImage.getGraphics();
 offScrGC.setColor(Color.lightGray); // default background
 // ...
 offScrGC.clearRect(0, 0, appWidth, appHeight);
 // frameNum is the framenummer; assume direction LEFT
 offScrGC.drawImage(image[i], -frameNum, 0, this);
 offScrGC.drawImage(image[i+1], appWidth - frameNum, 0,
 this);
 // now draw on screen
 g.drawImage(OffScrImage, 0, 0, this);
 // ...
}
```

## Working with URLs

- The **URL** class provides many constructors for creating URL objects. These constructors can all throw a **MalformedURLException** so it must be caught.
  - **public URL(String spec)**
  - **public URL(String protocol, String host, int port, String file)**
  - **public URL(URL context, String spec)**
- Example:
 

```
String param = getParameter("imagesource");
imageSource = new URL(getDocumentBase(), param + "/");
```

## Manually Controlling Layout (AWT)

- Java applets are containers: they can contain any number of components, which can themselves be containers. A layout manager can automate placement of components within a container, so that the applet can adapt itself to the size of the window it receives from the browser.
- An applet can [position and resize components](#) itself:
  - It first calls `setLayout(null)` to disable automatic positioning of components.
  - It adds components and calls `setBounds()` to position and resize them.

## Layout Managers

- Applets can place and size objects automatically:
  - `FlowLayout` works much like text is normally formatted.
  - `GridLayout` arranges components in a matrix.
  - `BorderLayout` places components in one of five areas (upon request): North, South, East, West or Center.
  - `CardLayout` treats the set of components as a stack of cards: only one card is visible at a time.
  - `GridBagLayout` is the most flexible and most used layout manager. It treats a container as a grid of cells, and a component can occupy more than one cell.

## GridBagLayout Example

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints constraints = new
GridBagConstraints();
constraints.fill = GridBagConstraints.BOTH;
setLayout(gridbag);
constraints.weightx = 1.0;
constraints.weighty = 1.0;
Button first = new Button("first");
gridbag.setConstraints(first, constraints);
add(first);
constraints.weightx = 2.0; // twice as wide
constraints.weighty = 3.0; // three times as high
Button second = new Button("second");
gridbag.setConstraints(second, constraints);
add(second);
```

## Adding (temporary) Windows

- The class `Frame` is used to create a [new toplevel window](#).
- `Menus` can appear on a `MenuBar` or can be a `PopupMenu`.
  - A `Frame` can have a `MenuBar` but an `Applet` cannot. (In `Swing` an `Applet` can have a `MenuBar`. Why the difference?)
- `Dialog boxes` are top-level windows that depend on another window. They can be modal.
  - An `Applet` cannot create a dialog box. Why not?

## Events

- In command-line user-interfaces and in most batch programs there is a *single input stream*.
- In graphical user-interfaces and process control there are typically *many input channels*.
- Handling multiple input streams is handled:
  - through *interrupts* in hardware; careful: an interrupt handler may be “blind” while handling an interrupt.
  - through *events* in software; careful: an event handler may be “blind” while handling an event. **We must ensure that the user-interface is never “blind”.**

## The Java 2 Event Model

- Based on event *sources* and *listeners*. An object that wishes to listen for a particular event must ask a source of such an event to be added to its list of listeners.
  - For example: an object that wishes to handle a button click must implement an `ActionListener` interface, and must call the button's `addActionListener` method to be notified of button clicks.
  - The “listening” object wants to pass a pointer to a method to be called to the event source, but this is not possible in Java. Therefore it must pass “itself” instead. The name of the event handlers must be standardized.

## Listeners

- Convention on names of listeners:
 

```
public class myMouse implements MouseListener {
 ...
 public void mouseClicked(MouseEvent e) {
 // handle the click
 }
 public void mousePressed(MouseEvent e) {
 // handle the press
 }
 public void mouseReleased(MouseEvent e) {
 // handle the release
 }
 public void mouseEntered(MouseEvent e) {
 // handle the enter event
 }
 ...
}
```

## Adapters

- Same convention as for listeners:
 

```
public class myMouse extends MouseAdapter {
 ...
 public void mouseClicked(MouseEvent e) {
 // handle the click
 }
 // re-implement only those methods we need
 ...
}
```

## Inner Classes

- Named or anonymous local classes, most useful for event handlers:
 

```
class MyClass extends Applet {
 ...
 someObject.addMouseListener(new MyAdapter());
 ...
 class MyAdapter extends MouseAdapter {
 public void mouseClicked(MouseEvent e) {
 ... //Event handler implementation goes here...
 }
 }
}
```

## Assignment: Room Thermostat

- Design, development, testing and evaluation of a graphical (computer) user-interface for a (room) thermostat.
  - There are two temperatures: day and night, which the user can set (to arbitrary values in the range of 5 to 30 degrees centigrade, up to 0.1 degrees).
  - There is one week-program. Each day may have different times for switching between day and night temperature. The week-program should be easy to specify and to review and update.
  - Each day may have up to five changes from day to night and five changes from night to day. Midnight is always an extra switch to the night temperature (unless this is the start of a day period). This number is fixed. Unused switches may not be transferred to another day.

## Assignment: Room Thermostat (cont.)

- It must be easy to override the current temperature temporarily, until the next programmed switch (or midnight). The temperature can be controlled by increments and decrements of 0.1 degrees.
- It must be easy to override the current temperature "permanently", that is until the user switches back to the week program.
- It must also be easy to set the day and time. The prototype should operate about 300 times faster than real time. (This means one second corresponds to 5 minutes.) The prototype must include a clock.
- The thermostat is intended for "normal" people, who are smart enough to use the timer on a VCR or microwave, but who are not computer specialists.

## Assignment: Room Thermostat (cont.)

- Implementation:
  - The application must interact with a **House** object, which is given (House.class, package House). The House object (thread) keeps track of the temperature changes, and provides an **int getTemperature()** call which returns 10 times the current room temperature in degrees centigrade. The House also offers a void **setTemperature(int)** method to set the desired temperature (in 1/10 degrees). **These methods do not work instantaneously. Their delay should not affect your thermostat!**
  - The heater needs some time to effectively heat up the room. There may be some hysteresis as well. When the desired temperature is lowered it takes some time for the room to cool down. The heating works well in the range of 5 to 30 degrees.

## Assignment: Room Thermostat (cont.)

- Required actions and documents:
  - You must hold a brainstorming session.
  - You must create a prototype as a **Java 2 applet** and an HTML page that contains the applet. (You may choose between AWT and Swing, but make sure you don't mix the two.) The prototype must work with Mozilla, Netscape and Internet Explorer client together with a Unix HTTP server.
  - You must hand in the brainstorming report on paper, and the prototype on a CD-ROM. The CD must contain one single ZIP archive or a gzipped tar archive. The CD and paper must carry the student id's and email addresses of all group members. The archive must include a readme file with installation instructions (even if trivial). The archive must include the House.class file from the course website.

## Assignment: Room Thermostat (cont.)

- Delivery and Evaluation
  - **deadline: March 2, 2007, 1.00pm**
  - **format: paper document + cd-rom**
  - **place: secretary (Lydia Jansen), building G, 1<sup>st</sup> floor**
  - We will e-mail to set up a meeting for evaluating your prototype. Three groups will work together during the evaluation of your assignment: one will carry out an experiment, while a second group observes that experiment. (You must be present to offer help in case the experiment goes terribly wrong.)