CS 127

An Introduction to Java[1]

Bruce M. Bolden
August 29, 2001

[1] ©Bruce M. Bolden, 1998–2001.

Contents

1 Introduction 1

2 Policies and Procedures 1
2.1 Grading 1
2.2 Academic Dishonesty 2
2.3 Computer Usage/Misuse 2

3 Tips for Success in this class 2
3.1 General 2
3.2 Programming Tips 2
3.3 Test Tips 3

4 Introduction to Java 4
4.1 What is Java? 4
4.2 Why Java? 4
4.3 How is Java different? 5
4.4 Brief History of Java 5
4.5 "Hello World" in C++ 6
4.6 "Hello World" in Java 7

5 Frequently used terms 8

6 Java API Packages 9

7 Java Growth 12

8 Compiling and Running a Java Program 13
8.1 A Simple Java Program 13
8.2 `javac` — The Java Compiler 14
8.3 `java` — The Java Interpreter 14
8.4 Java is Case Sensitive 14
8.5 Common Interpreter Problems 14

9 Variables 15
9.1 Variable Types 15
9.2 Variable Sizes 16
9.3 Variable Modifiers 17

9.4 Instance Variables . . . 17
9.5 Class Variables . . . 18
9.6 Local Variables . . . 18
9.7 Constants . . . 18

10 Introduction to Objects 19
10.1 Data Abstraction . . . 19
10.2 Encapsulation . . . 20
10.3 Objects . . . 20
10.4 Messages . . . 20
10.5 Methods . . . 21

11 Example: Geometric Objects 21
11.1 Objects in C: `structure` Technique . . . 21
11.1.1 Test Program . . . 22
11.2 Objects in C++: Class Technique . . . 23
11.2.1 Circle Class Interface . . . 23
11.2.2 Circle class implementation . . . 24
11.2.3 Test Program . . . 25
11.3 Objects in Java . . . 26
11.3.1 Circle Class . . . 26

12 Introduction to the `String` class 28
12.1 `String` Constructors . . . 28
12.2 `String` Usage . . . 30
12.3 `String` Methods . . . 30
12.4 Some Code from the `String` Class . . . 36

13 Classes and Methods 43
13.1 Instance Methods . . . 43
13.2 Class Methods . . . 43
13.3 `main()` Methods . . . 44
13.4 Overloaded Methods . . . 44
13.5 Overriding Methods . . . 44
13.6 Abstract Methods . . . 44
13.7 Final Methods . . . 45
13.8 Native Methods . . . 45
13.9 Sample Code . . . 46

14 Drawing using the AWT **49**
14.1 The CloseableFrame class 49
14.1.1 Inheritance 53
14.1.2 Using methods 53

15 Commonly used Graphics methods **53**
15.1 Graphics Coordinate System 55
15.2 Simple Graphics Program 56

16 Programming Assignments **59**
16.1 Program Development Model 59
16.2 Other design/implementation issues 60
16.3 Program Development Model Applied to the Checker Board Display Program 61
16.3.1 Specifications 61
16.3.2 Display Checker Board 61
16.3.3 Label Board Sides 61
16.3.4 Label Squares 61
16.3.5 Alternative Checker Board Displays 61
16.4 Program Development Model Applied to the LED Display Program 62
16.4.1 Specifications 62
16.4.2 LED Digit Class 62

17 Abstract Classes and Methods **63**
17.1 Abstract Rules 63
17.2 Sample Code 64

18 Interfaces **73**
18.1 ObjGeometry.java 73
18.2 Circle.java 74
18.3 TestGObject.java 75
18.4 Implementing Multiple Interfaces 77
18.5 Constants in Interfaces 77
18.6 Extending Interfaces 78

19 Vectors **80**
19.1 Vector API 80

19.2 Manipulating Vectors 81
19.2.1 Creating a Vector 81
19.2.2 Adding an element to a Vector 82
19.2.3 Vector Sizing Operations 82
19.2.4 Accessing Vector Elements 83
19.2.5 Inserting/Removing Elements in the Middle of a Vector 84
19.3 Example using the GObject and Vector Classes 85
19.3.1 Output written to `System.output` 90

20 Reflection 92
20.1 Using Reflection to Analyze the Capabilities of Classes 92
20.2 Example: Analyzing Classes using Reflection 94

21 Events 98
21.1 Event Handling Basics 98
21.1.1 Capturing Window Events 101
21.1.2 Adapter Classes 103
21.2 JDK 1.1 Event Hierarchy 104
21.2.1 Example: Button Selection 106

22 Events 110
22.1 Semantic and Low-Level Events 110
22.1.1 Event Handling Summary 112
22.1.2 Event Delegation Mechanism Review 113

23 Event Handling for Java Applications 115
23.1 Example: Java 115

24 Event Handling for Macintosh and Windows Applications 119
24.1 Example: Macintosh 119
24.2 Example: Windows 127

25 Input and Output in Java 129
25.1 Input and Output Streams 129
25.2 Input and Output Stream Processing 129
25.3 Input and Output Stream Hierarchies 129
25.3.1 Input Stream Hierarchy 130
25.3.2 Output Stream Hierarchy 131

25.3.3 Connected I/O Stream Hierarchy . . . 131
25.3.4 Reader Hierarchy . . . 132
25.3.5 Writer Hierarchy . . . 132
25.4 Reading and Writing Bytes . . . 133
25.4.1 Close Stream Objects . . . 133
25.5 More Stream Classes . . . 134
25.6 Stream Filters . . . 134
25.6.1 Stream Filter Example . . . 135
25.6.2 Buffered Stream Filters . . . 135
25.6.3 Example: Reading and Writing Bytes . . . 136
25.7 Reading and Writing Text . . . 137
25.7.1 Writing Text . . . 137
25.7.2 Writing To A Print Writer . . . 137
25.7.3 Reading Text . . . 138
25.7.4 Reading With A `BufferedReader` . . . 138
25.7.5 More Reading With A `BufferedReader` . . . 139
25.7.6 Example: Buffered Reading and Writing . . . 140
25.8 References . . . 141

26 Windows 142
26.1 Component Class . . . 143
26.1.1 Frame . . . 143
26.1.2 Canvas . . . 144
26.1.3 Panel . . . 144
26.1.4 Applet . . . 144
26.1.5 Dialog . . . 144
26.1.6 FileDialog . . . 145
26.1.7 ScrollPane . . . 145
26.2 Canvas . . . 146

27 Introduction to Components 153
27.1 Component Class Inheritance Hierarchy . . . 153
27.2 Component Example . . . 154
27.3 Component Example Source Code . . . 156

28 Components 161
28.1 Button . . . 161
28.2 Canvas . . . 161

28.3 Checkbox 161
28.4 Checkbox Groups (radio buttons) 162
28.5 Choice 163
28.6 Label 163
28.7 List 164
28.8 TextField 164
28.9 TextArea 164
28.10Example 165

29 Layout Managers 172
29.1 Flow Layout 172
29.2 Border Layout 175
29.3 Card Layout 178
29.4 Grid Layout 179
29.5 Grid Bag Layout 181
29.6 Custom Layout Managers 186

30 Applets 187
30.1 Writing an applet 187
30.2 More frequently used methods 188
30.3 Simple Applet 189
30.4 Event Handling 190
30.5 Applet using Card Layout 191

Preface

These notes represent the concepts you will be learning in CS 127. For some of you, these notes will be confusing initially—don't panic. We will discuss them in detail over the course of the semester.

Note: these notes are not a substitute for the textbook! They are not a substitute for attending lecture and taking notes either. Lecture is probably the most important part of this class, although most of your actual learning will take place while working on the programming assignments. A number of past programming assignments have been included at the end of this document.

The class web site (`http://www.cs.uidaho.edu/~bruceb/cs127/`) contains information that *supersedes any/all* information in this document. Please visit it regularly to view the latest information about this class.

These notes are dynamic—they are under constant revision. If you find any errors, please let me know so that I can correct them for the next class. Speaking of errors, I have made every effort to organize the source code so that it will be easy for you to read and add additional notes if we discuss the example in class.

Now, let's get on with it!

Bruce
`bruceb@cs.uidaho.edu`

1 Introduction

Bruce Bolden

Contact Information:

email:	`bruceb@cs.uidaho.edu`
web:	`http://www.cs.uidaho.edu/~bruceb/`
office:	JEB 220
phone:	Don't bother

The class web site contains information that *supersedes any/all* information in this document. Please visit it regularly to view the latest information about this class.

2 Policies and Procedures

2.1 Grading

There will be numerous programming assignments in this class. It is expected that students will do their own work on all components of the programs—unless otherwise specified.

Quizzes will normally be given on Wednesdays on the material covered since the last quiz. Knowledge of material presented in this class is cumulative!

The final grade will be calculated based on a weighted sum of the points accumulated in each of the categories[2]:

Programs	30
Quizzes	20
Tests	30
Final Exam	20

The course will be graded on the basis of 90% and above is an A, 80%–89% a B, 70%–79% a C, etc.

[2]See web site for current weightings

2.2 Academic Dishonesty

Cheating on exams or homework will be heavily penalized.

2.3 Computer Usage/Misuse

Misuse of computers and files is a felony in the state of Idaho! See the University of Idaho Computer Use Policy document available from Computer Services for details.

3 Tips for Success in this class

3.1 General

- Attend class regularly.
- Check the class web site (daily).
- If you have a question, do not wait to ask it.
- Read/Review material regularly.
- Review programs after completion.
- Organize your notes and other material for the class.

3.2 Programming Tips

- Read/Review the assignment as soon as possible.
- Think about how you might solve the problem.
- Design a solution to the problem.
- Implement your solution.
- Allow time for problems.
- Review your assignment to make sure that it satisfies the assignment.

3.3 Test Tips

- Keep up with the material as it is discussed.
- Start studying early.
- Read/Review material regularly.
- Rewrite and think about the material as necessary.
- Read all problems on the test before starting. Make sure you understand the problem before starting.
- Work as quickly as possible, without going so fast that you make careless errors.
- Review your answers.
- Relax!

4 Introduction to Java

Some possible questions:

- What is Java?
- Why Java?
- What about C++?
- What about programming for the Internet?
- How does Java differ from other programming languages?

4.1 What is Java?

- An Object-Oriented programming language.
- An Internet programming language.
- A concurrent programming language.
- A distributed programming language.
- A strongly-typed programming language.

4.2 Why Java?

- Designed from the ground up—objects in C++ were an addition to C.
- Multilingual.
- Designed to support:
  - Concurrent programming
  - Distributed programming
  - Internet programming

4.3 How is Java different?

- File and class name must be the same.
- JVM interprets byte code.
- Not C++!
- Not Javascript.
- Very rich, built-in graphics and GUI capabilities.
- All *objects* are passed by reference.

4.4 Brief History of Java

- Fairly new language.
- Wide-spread availability/usage.
  - Personal Computers
  - Super Computers
- Consistent Implementations
- Still evolving (e.g., JDK 1.3 last Fall).
- Based upon a *virtual machine* (UCSD Pascal mid '70's)

4.5 “Hello World” in C++

```
/*  hello1.cpp
 *
 *  Sample program that displays a message.
 *
 *  Bruce M. Bolden
 *  August 18, 1997
 */

#include <iostream.h>

int main()
{
       //  write message to standard output
    cout << "Hello, World!" << endl;
}
```

After this program is compiled (successfully) and run, it will display the message *Hello, World!* on the screen (standard output device).

4.6 "Hello World" in Java

```
/*  Hello.java
 *
 *  Sample program that displays a message.
 *
 *  Bruce M. Bolden
 *  December 22, 2000
 */

public class Hello
{
  public static void main( String[] args )
  {
      System.out.println( "Hello!" );
  }
}
```

After this program is compiled (successfully) and run, it will display the message *Hello, World!* in the Java console window (standard output device).

Items of note:

- Program is compiled
- Comments
- No `#include` statments
- `System.out.println`
- Usage of double quotes
- Semi-colon ends statement

Special Characters:

> Recall that `\n` is the escape code for newline. It may be thought of as a *return* at the end of a line.
>
> Similarly, `\t` is the escape code for a tab. Note a tab is not a defined number of spaces and is typically *editor* dependent.

5 Frequently used terms

The world is filled with TLAs (three letter acronyms).

ABI	Application Binary Interface
API	Application Programming Interface
FAQ	Frequently Asked Question(s)
GUI	Graphical User Interface
IDE	Integrated Development Environment
OOP	Object-Oriented Programming
SDK	Software Development Kit
CGI	Common Gateway Interface
HTML	Hypertext Markup Language
URL	Uniform Resource Location
WWW	World Wide Web
AWT	Abstract Window Toolkit
BDK	Bean Development Kit
JDK	Java Development Kit
JVM	Java Virtual Machine
AFC	Application Foundation Classes (Microsoft)
IFC	Internet Foundation Classes (Netscape)
JFC	Java Foundation Classes (JavaSoft/Netscape)
JGL	Java Generic Library (Objectspace)
	Generic Collection Library for Java
JAR	Java Archive
JIT	Just In Time (compiler)
JLF	Java Look and Feel
JRE	Java Runtime Environment
JNI	Java Native Interface
NMI	Native Method Interface
RMI	Remote Method Invocation

6 Java API Packages

There are 538 public classes within the 24 core packages in the JDK 1.1.
Sources:
Package names from *The Java Class Libraries Poster, Java 1.1* From a link from the *Java Class Libraries: Second Edition*, Volume 2, Patrick Chan and Rosanna Lee.
Numbers from *Just Java 1.1*, Third Edition, Peter van der Linden Prentice-Hall.

1. java.applet (4 classes)
2. java.awt (141 classes)
3. java.awt.datatransfer
4. java.awt.event
5. java.awt.image
6. java.awt.peer
7. java.beans (23 classes)
8. java.io (71 classes)
9. java.lang (77 classes)
10. java.lang.reflect
11. java.math (2 classes)
12. java.net (31 classes)
13. java.rmi (48 classes)
14. java.rmi.dgc
15. java.rmi.registry
16. java.rmi.server
17. java.security (40 classes)

18. java.security.acl
19. java.security.interfaces
20. java.sql (17 classes)
21. java.text (41 classes)
22. java.text.resources
23. java.util (44 classes)
24. java.util.zip

The packages of most interest to us will be:

1. java.applet
2. java.awt
3. java.awt.event
4. java.awt.image
5. java.io
6. java.lang
7. java.math
8. java.util

We may see some items from:

1. java.awt.datatransfer
2. java.beans
3. java.lang.reflect
4. java.net
5. java.text

6. java.util.zip

It is unlikely we will look at:

1. java.awt.peer
2. java.rmi
3. java.rmi.dgc
4. java.rmi.registry
5. java.rmi.server
6. java.security
7. java.security.acl
8. java.security.interfaces
9. java.sql
10. java.text.resources

7 Java Growth

	JDK 1.02	JDK 1.1	JDK 1.2b3
Packages	8	23	62
Classes and Interfaces	212	504	1592
Classes	172	391	1287
`abstract`	21	58	194
`final`	20	38	160
`protected`			104
`abstract protected`			1
Interfaces	40	113	305
Members	2165	5478	18837
Fields	261	926	3107
`static`	3	4	71
`final`		2	20
`static final`	167	764	2018
`protected`	50	111	861
Constructors	319	701	2095
`protected`	6	43	189
Methods	1545	3851	13635
`abstract`	102	202	868
`static`	149	360	987
`final`	140	207	329
`static final`	1	24	57
`protected`	78	191	1192

Source:
Numbers from:
The Java Developers Almanac, 1998, Patrick Chan, Addison-Wesley, 1998.

8 Compiling and Running a Java Program

The Java Compilation Process

8.1 A Simple Java Program

```
//  Hello.java
public class Hello {
  public static void main(String[] args)
  {
       System.out.println( "Hello!" );
  }
}
```

To compile this source file, type:

```
javac Hello.java
```

To run the program, type:

```
java  Hello
```

8.2 `javac` — The Java Compiler

`javac` is the Java compiler. It compiles the file(s) that follow it on the command line. If there are no errors, it generates class file(s). So, for this example `Hello.class` is generated when `Hello.java` is compiled (if there are no syntax or other compilation errors).

8.3 `java` — The Java Interpreter

`java` is the Java interpreter. It interprets the bytecodes in the class file generated by the compiler. It starts execution at the `main()` function within the specified class. **Note:** If a `main()` function is not found an error will occur.

Both `javac` and `java` have command line options. Check the documentation for details.

8.4 Java is Case Sensitive

Java is case sensitive! DOS is not. Use care when entering commands. If you encounter an error during compilation or execution, check the file names, class names, parameter names, etc., for capitalization.

8.5 Common Interpreter Problems

Can't find class
Check the argument passed to the interpreter. The argument is the *name of the class* that you want to use, *not* the filename.

main method is not defined
Make sure that the class has a main method.

Changes did not take effect
It may be necessary to delete the class file generated by a previous compilation.

9 Variables

There are three kinds of variables in Java:

1. *Instance* variables — variables that hold data for an object (a class instance).

2. *Class* variables — variables that hold data that can be shared among all instances of a class.

3. *Local* variables — variables that are only used in a block of code (e.g., inside methods, loops, static initializers).

9.1 Variable Types

A variable's type specifies the kinds of values that may be stored in the variable. A variable can hold any of the *primitive* types in Java—`boolean, byte, char, short, int, long, float, double`) or an instance of a class. A variable of type `Object` can hold an instance of any class, since all classes are subclasses of `Object`.

Sample variable declarations:

```
int  i;
boolean done = false;
Color  c;
String s = new String( "Hi" );
Frame  f = new Frame();
```

9.2 Variable Sizes

The size (in bits) and range of the basic variable types is given in the following table.

Type	**Contains**	**Size**	**Range**
boolean	`true` or `false`	1 bit	
byte	signed integer	8 bits	-128 .. 127
char	Unicode character	16 bits	`\u0000 .. \uFFFF`
short	signed integer	16 bits	-32768 .. 32767
int	signed integer	32 bits	-2147483648 .. 2147483647
long	signed integer	64 bits	-9223372036854775808 .. 9223372036854775807
float	IEEE 754 floating point	32 bits	-3.402E-38 .. 3.402E+38
double	IEEE 754 floating point	64 bits	-1.797E±308 .. 1.797E±308

9.3 Variable Modifiers

Some of the same modifiers used on methods can also be used on variables.

Modifier	The variable ...
`public`	can be accessed by any class
`private`	can be accessed only by methods within the same class
`protected`	can be accessed only by subclasses and classes in the same class
`static`	is a class variable
`final`	cannot be changed

9.4 Instance Variables

Instance variables hold the data for an instance of a class.

The initial value of an instance variable can be specified explicitly. For example,

```
int  i = 1;
```

initializes `i` to 1.

If the initial value of an instance variable is not specified, Java assigns a default value. The following table shows the default values assigned for each type of variable.

Type	**Initial value**
`boolean`	`false`
`byte`	`0`
`char`	`'\u0000'`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
others	`null`

`null` is a special value that can be assigned to any variable that holds an object. It is guaranteed to be distinct from any other value.

9.5 Class Variables

Class variables hold data that is shared among all the instances of a class. A class variable is declared using the `static` modifier. For example,

```
private static int MAX_SQUARES = 8;
```

Class variables are initialized the same way as instance variables.

9.6 Local Variables

Local variables in Java are similar to local variables in other languages. Local variables can occur anywhere inside a method or block (even a static initialization block).

Local variables are not initialized by Java—no default initial value.

9.7 Constants

Constants in Java are different than in many other programming languages. The modifier `final` is used to specify that the value of a variable will not change.

[Note: Before JDK 1.1, Java did not allow a local variable to be declared `final`.]

10 Introduction to Objects

What are objects?

Software objects represent an abstraction of some reality.

What is Object-Oriented Programming (OOP)?

OOP is a way of thinking about problem solving and a method of software organization and construction.

In object-oriented programming, each managed piece of state is callend an *object*. An object consists of several stored quantities, called its *fields*, with associate *methods* (functions) that have access to the fields.

To facilitate the sharing of methods, object oriented programming systems typically provide *classes* which are structures that specify the fields and methods of each such object. Each object is created as an *instance* of some class.

10.1 Data Abstraction

Data abstraction associates some underlying data type with a set of operations that may be performed on the data type. It is not necessary for a user (programmmer) of the data type to know how the type is represented (i.e., how the information in the type is stored) but only how the information can be manipulated.

Data abstraction derives its strength from the separation between the operations that may be performed on the underlying data and the internal representation of these data.

10.2 Encapsulation

The combination of the underlying data and a set of operations that define the data type is called encapsulation. The internal representation of the data is encapsulated (hidden) but can be manipulated by the specified operations.

10.3 Objects

OOP is based on the concept of objects. A software object represents an abstraction of some reality.

OOP is also based on the concept of sending messages to objects. Messages can *modify* or *return* information about the internal state of an object.

The behavior of an object is codified in a class description. The object is said to be an instance of the class that describes its behavior. The class description specifies the internal state of the object and defines the types of messages that may be sent to all its instances.

An object's *value* or information content is given by its internal state. This internal state is defined in terms of one or more fields. Each field holds a portion of the information content of the object.

10.4 Messages

Messages are sent to or invoked on objects. In most object-oriented languages the syntax used to do this is:

```
anObject.aMessage
```

In Java (and C++), the syntax is:

```
anObject.aMessage()
```

Note: A message may have one or more arguments.

10.5 Methods

A method is a function or procedure that defines an action associated with a message. It is defined as part of the class description. When a message is invoked on an object, the details of the operation are specified by the corresponding method.

11 Example: Geometric Objects

Basic geometric objects:

- Circle
- Square
- Rectangle

How can we define them?

11.1 Objects in C: `structure` Technique

A circle structure:

```
struct circle {
   double radius;
};

typedef struct circle Circle;
```

11.1.1 Test Program

```
#include <iostream.h>

void SetRadius( Circle& c, double r );
void ShowRadius( Circle c );

main()
{
    Circle c1;

    c1.radius = 2.0;
    ShowRadius( c1 );
    SetRadius( c1, 3.2 );
    ShowRadius( c1 );

    return 0;
}

void SetRadius( Circle& c, double r )
{
    c.radius = r;
}

void ShowRadius( Circle c )
{
    cout << c.radius << endl;
}
```

11.2 Objects in C++: Class Technique

11.2.1 Circle Class Interface

```
/*  Circle.h
 *
 *  Circle class interface
 *
 *  Bruce M. Bolden
 *  January 20, 2001
 */

class Circle
{
public:
            //  constructors
        Circle();
        Circle( double r );

  void  SetRadius( double r );
  void  ShowRadius();

private:
  double radius;
};
```

11.2.2 Circle class implementation

```
/*  Circle.cpp
 *
 *  Circle class implementation
 *
 *  Bruce M. Bolden
 *  January 20, 2001
 */

#include <iostream.h>
#include "Circle.h"

   // Constructors
Circle::Circle() {
    radius = 1.0;
}

Circle::Circle( double r ) {
    radius = r;
}

void Circle::SetRadius( double r ) {
    radius = r;
}

void Circle::ShowRadius() {
    cout << radius << endl;
}

/*  end of Circle class implementation */
```

11.2.3 Test Program

```
/*  TestCircle.cpp
 *
 *  Circle class test program.
 *
 *  Bruce M. Bolden
 *  January 20, 2001
 */

#include <iostream.h>

#include "Circle.h"

int main()
{
    Circle c1;
    Circle c2( 3.0 );

    cout << "The radius of c1 is: ";
    c1.ShowRadius();

    cout << "The radius of c2 is: ";
    c2.ShowRadius();

    c1.SetRadius( 2.1 );
    cout << "The radius of c1 is: ";
    c1.ShowRadius();

    return EXIT_SUCCESS;
}
```

11.3 Objects in Java

11.3.1 Circle Class

```
/*  Circle.java
 *
 *  Bruce M. Bolden
 *  January 20, 2001
 */

public class Circle
{
    double radius;

        // Constructors
    public Circle() {
        radius = 1.0;
    }

    public Circle( double r ) {
        radius = r;
    }

    public void setRadius( double r ) {
        radius = r;
    }

    public void showRadius() {
        System.out.println( radius );
    }

    public static void main( String [] args )
```

```
    {
        Circle c1 = new Circle();
        Circle c2 = new Circle( 5.31 );

        System.out.print( "Radius of c1: " );
        c1.showRadius();
        System.out.print( "Radius of c2: " );
        c2.showRadius();

        c1.setRadius( 2.1 );
        System.out.print( "Radius of c1: " );
        c1.showRadius();
    }
}
```

12 Introduction to the `String` class

What are `Strings`?

A class that hold character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of the String class.

How are `Strings` implemented?

The `String` class stores a string in an array of characters.

12.1 `String` Constructors

1. `String()` Allocates a new `String` containing no characters. Initializes a newly created String object so that it represents an empty character sequence.

    ```
    public String()
    {
        value = new char[0];
    }
    ```

2. `String( byte[] bytes )`
   Construct a new String by converting the specified array of bytes using the platform's default character encoding.

3. `String( byte[] ascii, int hibyte )`
   Deprecated.This method does not properly convert bytes into characters. As of JDK1.1, the preferred way to do this is via the String constructors that take a character-encoding name or that use the platform's default encoding.

4. `String( byte[] bytes, int offset, int length )`
   Construct a new String by converting the specified subarray of bytes using the platform's default character encoding.

5. `String( byte[] ascii, int hibyte, int offset, int count )`
   Deprecated.This method does not properly convert bytes into characters. As of JDK1.1, the preferred way to do this is via the String constructors that take a character-encoding name or that use the platform's default encoding.

6. `String( byte[] bytes, int offset, int length, String enc )`
   Construct a new String by converting the specified subarray of bytes using the specified character encoding.

7. `String( byte[] bytes, String enc )`
   Construct a new String by converting the specified array of bytes using the specified character encoding.

8. `String( char[] value )`
   Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

9. `String( char[] value, int offset, int count )`
   Allocates a new String that contains characters from a subarray of the character array argument.

10. `String( String value )`
    Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

11. `String( StringBuffer buffer )`
    Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

12.2 `String` Usage

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

12.3 `String` Methods

The `String` class includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

1. `char charAt( int index )`
   Returns the character at the specified index.

2. `int compareTo( Object o )`
   Compares this String to another Object.

3. `int compareTo( String anotherString )`
   Compares two strings lexicographically.

4. `int compareToIgnoreCase( String str)`
   Compares two strings lexicographically, ignoring case considerations.

5. `String concat( String str )`
   Concatenates the specified string to the end of this string.

6. `staticString copyValueOf( char[] data )`
   Returns a String that is equivalent to the specified character array.

7. `staticString copyValueOf( char[] data, int offset, int count )`
   Returns a String that is equivalent to the specified character array.

8. `boolean endsWith( String suffix )`
   Tests if this string ends with the specified suffix.

9. `boolean equals( Object anObject )`
   Compares this string to the specified object.

10. `boolean equalsIgnoreCase( String anotherString )`
    Compares this String to another String, ignoring case considerations.

11. `byte[] getBytes()`
    Convert this String into bytes according to the platform's default character encoding, storing the result into a new byte array.

12. `void getBytes( int srcBegin, int srcEnd, byte[] dst, int dstBegin )`
    Deprecated.This method does not properly convert characters into bytes. As of JDK1.1, the preferred way to do this is via the getBytes(String enc) method, which takes a character-encoding name, or the getBytes() method, which uses the platform's default encoding.

13. `byte[] getBytes( String enc )`
    Convert this String into bytes according to the specified character encoding, storing the result into a new byte array.

14. `void getChars( int srcBegin, int srcEnd, char[] dst, int dstBegin )`
    Copies characters from this string into the destination character array.

15. `int hashCode()`
    Returns a hashcode for this string.

16. `int indexOf( int ch )`
    Returns the index within this string of the first occurrence of the specified character.

17. `int indexOf( int ch, int fromIndex )`
    Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

18. `int indexOf( String str )`
    Returns the index within this string of the first occurrence of the specified substring.

19. `int indexOf( String str, int fromIndex )`
    Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

20. `String intern()`
    Returns a canonical representation for the string object.

21. `int lastIndexOf( int ch )`
    Returns the index within this string of the last occurrence of the specified character.

22. `int lastIndexOf( int ch, int fromIndex )`
    Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

23. `int lastIndexOf( String str )`
    Returns the index within this string of the rightmost occurrence of the specified substring.

24. `int lastIndexOf( String str, int fromIndex )`
    Returns the index within this string of the last occurrence of the specified substring.

25. `int length()`
    Returns the length of this string.

26. `boolean regionMatches( boolean ignoreCase, int toffset, String other, int ooffset, int len )`
    Tests if two string regions are equal.

27. `boolean regionMatches( int toffset, String other, int ooffset, int len )`
    Tests if two string regions are equal.

28. `String replace( char oldChar, char newChar )`
    Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

29. `boolean startsWith( String prefix )`
    Tests if this string starts with the specified prefix.

30. `boolean startsWith( String prefix, int toffset )`
    Tests if this string starts with the specified prefix beginning at a specified index.

31. `String substring( int beginIndex )`
    Returns a new string that is a substring of this string.

32. `String substring( int beginIndex, int endIndex )`
    Returns a new string that is a substring of this string.

33. `char[] toCharArray()`
    Converts this string to a new character array.

34. `String toLowerCase()`
    Converts all of the characters in this String to lower case using the rules of the default locale, which is returned by Locale.getDefault.

35. `String toLowerCase( Locale locale )`
    Converts all of the characters in this String to lower case using the rules of the given locale.

36. `String toString()`
    This object (which is already a string!) is itself returned.

37. `String toUpperCase()`
    Converts all of the characters in this String to upper case using the rules of the default locale, which is returned by Locale.getDefault.

38. `String toUpperCase( Locale locale )`
    Converts all of the characters in this String to upper case using the rules of the given locale.

39. `String trim()`
    Removes white space from both ends of this string.

40. `static String valueOf( boolean b )`
    Returns the string representation of the boolean argument.

41. `static String valueOf( char c )`
    Returns the string representation of the char argument.

42. `static String valueOf( char[] data )`
    Returns the string representation of the char array argument.

43. `static String valueOf( char[] data, int offset, int count )`
    Returns the string representation of a specific subarray of the char array argument.

44. `static String valueOf( double d )`
    Returns the string representation of the double argument.

45. `static String valueOf( float f )`
    Returns the string representation of the float argument.

46. `static String valueOf( int i )`
    Returns the string representation of the int argument.

47. `static String valueOf( long l )`
    Returns the string representation of the long argument.

48. `static String valueOf( Object obj )`
    Returns the string representation of the Object argument.

12.4 Some Code from the `String` Class

```
/*
 * @(#)String.java 1.87 99/01/22
 *
 * Copyright 1995-1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

package java.lang;

import java.util.Hashtable;
import java.util.Locale;
import sun.io.ByteToCharConverter;
import sun.io.CharToByteConverter;
import java.io.CharConversionException;
import java.io.UnsupportedEncodingException;

/**
```

```
* The <code>String</code> class represents character
* strings. All string literals in Java programs, such
* as <code>"abc"</code>, are implemented as instances
* of this class.
* <p>
* Strings are constant; their values cannot be changed
* after they are created.  String buffers support mutable
* strings.  Because String objects are immutable they can
* be shared.  For example:
* <p><blockquote><pre>
*     String str = "abc";
* </pre></blockquote><p>
* is equivalent to:
* <p><blockquote><pre>
*     char data[] = {'a', 'b', 'c'};
*     String str = new String(data);
* </pre></blockquote><p>
* Here are some more examples of how strings can be used:
* <p><blockquote><pre>
*     System.out.println("abc");
*     String cde = "cde";
*     System.out.println("abc" + cde);
*     String c = "abc".substring(2,3);
*     String d = cde.substring(1, 2);
* </pre></blockquote>
* <p>

* The class <code>String</code> includes methods for
* examining individual characters of the sequence,
* for comparing strings, for searching strings, for
* extracting substrings, and for creating a copy of
```

```
 * a string with all characters translated to uppercase
 * or to lowercase.
 * <p>
 * The Java language provides special support for the
 * string concatentation operator ( + ), and
 * for conversion of other objects to strings.  String
 * concatenation is implemented through the
 * <code>StringBuffer</code> class and its
 * <code>append</code> method.  String conversions are
 * implemented through the method <code>toString</code>,
 * defined by <code>Object</code> and inherited by all
 * classes in Java. For additional information on string
 * concatenation and conversion, see Gosling, Joy, and
 * Steele, <i>The Java Language Specification</i>.

 *
 * @author  Lee Boynton
 * @author  Arthur van Hoff
 * @version 1.87, 01/22/99
 * @see     java.lang.Object#toString()
 * @see     java.lang.StringBuffer
 * @see     java.lang.StringBuffer#append(boolean)
 * @see     java.lang.StringBuffer#append(char)
 * @see     java.lang.StringBuffer#append(char[])
 * @see     java.lang.StringBuffer#append(char[], int, int)
 * @see     java.lang.StringBuffer#append(double)
 * @see     java.lang.StringBuffer#append(float)
 * @see     java.lang.StringBuffer#append(int)
 * @see     java.lang.StringBuffer#append(long)
 * @see     java.lang.StringBuffer#append(java.lang.Object)
 * @see     java.lang.StringBuffer#append(java.lang.String)
```

```
 * @since   JDK1.0
 */

public final
class String implements java.io.Serializable {
    /** The value is used for character storage. */
    private char value[];

    /** The offset is the first index of the storage
        that is used. */
    private int offset;

    /** The count is the number of characters in the
        String. */
    private int count;

    /** use serialVersionUID from JDK 1.0.2 for
        interoperability */
    private static final long
            serialVersionUID = -6849794470754667710L;

    /**
      * Allocates a new <code>String</code> containing
      * no characters.  */

    public String() {
       value = new char[0];
    }

/**
  * Allocates a new string that contains the same
  * sequence of characters as the string argument.
```

```
 *
 * @param   value  a <code>String</code>.
 */
 public String(String value) {
     count = value.length();
     this.value = new char[count];
     value.getChars(0, count, this.value, 0);
 }

/**
  * Copies characters from this string into the destination
  * character array.
  * <p>
  * The first character to be copied is at index <code>srcBegin</code>;
  * the last character to be copied is at index <code>srcEnd-1</code>
  * (thus the total number of characters to be copied is
  * <code>srcEnd-srcBegin</code>). The characters are copied into the
  * subarray of <code>dst</code> starting at index <code>dstBegin</code>
  * and ending at index:
  * <p><blockquote><pre>
  *     dstbegin + (srcEnd-srcBegin) - 1
  * </pre></blockquote>
  *
  * @param      srcBegin   index of the first character in the string
  *                        to copy.
  * @param      srcEnd     index after the last character in the string
  *                        to copy.
  * @param      dst        the destination array.
  * @param      dstBegin   the start offset in the destination array.
  * @exception StringIndexOutOfBoundsException If srcBegin or srcEnd is out
  *              of range, or if srcBegin is greater than the srcEnd.
  */
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {
   if (srcBegin < 0) {
       throw new StringIndexOutOfBoundsException(srcBegin);
   }
```

```
    if (srcEnd > count) {
        throw new StringIndexOutOfBoundsException(srcEnd);
    }
    if (srcBegin > srcEnd) {
       throw new StringIndexOutOfBoundsException(srcEnd - srcBegin);
    }
    System.arraycopy(value, offset + srcBegin, dst, dstBegin, srcEnd - srcBegin);
}

/**
  * Converts this string to a new character array.
  *
  * @return  a newly allocated character array whose
  *          length is the length of this string and
  *          whose contents are initialized to contain
  *          the character sequence represented by this
  *          string.
  */
public char[] toCharArray() {
    int max = length();
    char result[] = new char[max];
    getChars(0, max, result, 0);
    return result;
}

/**
  * Returns the length of this string.
  * The length is equal to the number of 16-bit
  * Unicode characters in the string.
  *
  * @return  the length of the sequence of characters
  *          represented by this object.
  */
```

```
public int length() {
    return count;
}
```

13 Classes and Methods

The fundamental structure in Java is a class. Class definitions in Java are analagous in many ways to classes in C++. The instance variables and methods of Java are very similar to the member variables and functions of C++. However, there are some very important differences!

An *object* (an instance of a Java class) can have both variables and methods associated with it. Java has nothing that corresponds to C/C++'s global variables or free-standing functions (e.g., `sin()`, `strcmp()`, `malloc()`, etc.)

13.1 Instance Methods

In Java, an instance method is a method associated with a specific instance of a class. When an instance method is invoked (called), it has access to the data contained in the object.

13.2 Class Methods

Class Methods are associated with a class as a whole—not a particular instance. Class methods are declared `static`.

`static` *type* `method_name(` *args* `)`

Static methods are invoked by using the class name followed by the method name:

`class_name.method_name(` *args* `)`

All class methods are implicitly *final* methods—they cannot be overridden.

13.3 `main()` Methods

Any class can contain a `main()` method. This is very useful for testing. The only `main()` that is recognized is the one in the class used as an argument to the `java` interpreter.

13.4 Overloaded Methods

Methods may be overloaded. This is very useful when writing constructors. An example of overloading a constructor is illustrated in the sample code.

13.5 Overriding Methods

A class can be a subclass of another class (*inheritance*). All variables and methods of the super (or parent) class are also also available to the subclass. (There are some exceptions to this, as we shall see).

To override a method, the new method must have the same *signature* as the method in it's superclass. (The method name, argument types, and return type are referred to as the signature of the method.)

To extend a method, Java provides the `super` reserved word to invoke the superclass method from within the new method.

13.6 Abstract Methods

The interface of a method can be specified without defining the actual method. This can be very useful when you know that classes will have to provide a particular operation.

13.7 Final Methods

Final methods cannot be overridden by methods in subclasses. When a method is declared as `final`, this means that the implementation will not change. It also allows the Java compiler to perform some amount of optimization.

13.8 Native Methods

Not all code can/must be written in Java. Java is platform independent and it is sometimes necessary to write platform-specific code. We will not be using `native` methods in this class.

13.9 Sample Code

The following program calculates and displays the area of a circle and a square (well known geometric objects).

Output:

```
    circle: 3.141592653589793
    circle: 12.566370614359172
    square: 4.0
GObject@8fb55a
```

```
/*  TestGObject.java
 *
 *  Bruce M. Bolden
 *  January 31, 2001
 *  Revision:  3
 *  Original:  January 26, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.io.*;

public class TestGObject
{
    public static void main( String[] args )
    {
        GObject circle  = new GObject( GObject.circle );
        GObject circle2 = new GObject( GObject.circle, 2 );
        GObject square  = new GObject( GObject.square, 2 );

        circle.showArea();
        circle2.showArea();
        square.showArea();

        System.out.println( circle.toString() );
    }
}
```

```
/*  GObject.java
 *
 *  Basic properties of geometric objects.
 *
 *  Bruce M. Bolden
 *  January 26, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.io.*;

public class GObject
{
        //  Object type codes
    final static int circle = 1;
    final static int square = 2;

    int oType;    //  object type
    int iDim;     //  key dimension
    double area;  //  area of the object

        //  constructors
    GObject( int type ) {
        this( type, 1 );
    }

    GObject( int type, int dim ) {
        oType = type;
        iDim = dim;
    }

        //  area display method
    public void showArea()
    {
        calcArea();

        if( oType == GObject.circle )
        {
```

```
            System.out.println( "\tcircle: " + area );
        }
        else if( oType == GObject.square )
        {
            System.out.println( "\tsquare: " + area );
        }
    }

         //  area calculation method
    private void calcArea()
    {
        if( oType == GObject.circle )
        {
            area = Math.PI * iDim * iDim;
        }
        else if( this.oType == GObject.square )
        {
            area = iDim * iDim;
        }
    }

    /**  test function
     */
    public static void main( String[] args )
    {
        GObject circle = new GObject( GObject.circle, 2 );
        GObject square = new GObject( GObject.square, 4 );

        circle.showArea();
        square.showArea();
    }
}
```

14 Drawing using the AWT

In the AWT, a top-level window is called a *frame*. Frames can contain other user interface components such as buttons and text fields (as we shall see later).

The default frame class (`Frame`) typically needs to be extended to be of any real use. Why?

- The default frame is sized at 0 by 0 pixels and is placed in the upper left corner of the screen.
- No event handling.

At this point, the first item is much more important than the second. [Aside: This has been the source of problems for many people starting to program in Java—they forget to resize their frame and nothing is visible]

Horstmann and Cornell, *Core Java, Volume I*, provide a useful class, `CloseableFrame`, that takes care of the shortcomings of the standard `Frame` class. `CloseableFrame` is part of the `corejava` *package*. The `CloseableFrame` class is described in the next section.

14.1 The CloseableFrame class

A `CloseableFrame` object starts out:

- At the default location for a frame (top left corner).
- With a size of 300 by 200 pixels.
- Displaying a title bar that contains the name of the class.

How is this done? By defining the `CloseableFrame` class as follows[3]:

```
//  CloseableFrame.java
package corejava;

import java.awt.*;
import java.awt.event.*;

public class CloseableFrame extends Frame
{
   public CloseableFrame()
   {
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e)
            { System.exit(0); }
      } );
      setSize(300, 200);
      setTitle(getClass().getName());
   }
}
```

Important statements in the `CloseableFrame` class:

- `import` statements precede class definition.
- `public class CloseableFrame extends Frame` says that the class, `CloseableFrame`, inherits its properties and behavior from the `Frame` class. The `Frame` class also inherits from its parent class (`Window`). The complete inheritance hierarchy for `CloseableFrame` is illustrated in the following figure.

[3]Horstmann and Cornell, Core Java, Volume I, Prentice Hall, 1997

- `public CloseableFrame()` is the *constructor* for the `CloseableFrame` class. [C++ programmers: Note that there is no return type.]
- `addWindowListener(new WindowAdapter() ...)` defines what happens if a Window close event is sent to the frame.
- `setSize(300, 200);` resizes the frame.
- `setTitle(getClass().getName());` sets the title bar to the same name as the class (this can be redefined later using `setTitle()`).

`CloseableFrame` Inheritance Hierarchy

As previously stated, `CloseableFrame` inherits methods from *all* of the classes above it.

- What does this mean?
- How do we use these methods?

14.1.1 Inheritance

All variables and methods in a `Frame` object are also available in a `CloseableFrame` object. We will discuss this in more detail later.

14.1.2 Using methods

To use a method, we must create an object (unless the methods are in the same class). This will be illustrated in the example program in the next section.

15 Commonly used Graphics methods

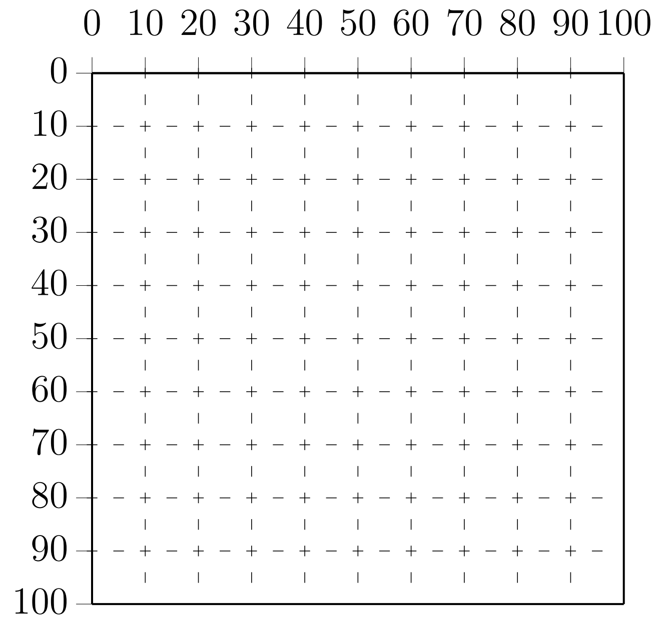
A reduced version of the `Graphics` class is shown below. The methods listed are the most useful for basic drawing.

```
public  abstract class Graphics extends Object {

    public  abstract  void translate( int x, int y ) ;

    public  abstract  void setColor( Color c ) ;

    public  abstract  void setFont( Font font ) ;

    public  abstract  void drawLine( int x1, int y1, int x2, int y2 ) ;

    public  abstract  void fillRect( int x, int y, int width, int height ) ;

    public  void drawRect( int x, int y, int width, int height ) ;

    public  abstract  void clearRect( int x, int y, int width, int height ) ;

    public  abstract  void drawRoundRect( int x, int y, int width, int height,
                                          int arcWidth, int arcHeight ) ;
```

```
    public  abstract  void fillRoundRect( int x, int y, int width, int height,
                                          int arcWidth, int arcHeight ) ;

    public  void draw3DRect( int x, int y, int width, int height,
                             boolean raised ) ;

    public  void fill3DRect( int x, int y, int width, int height,
                             boolean raised ) ;

    public  abstract  void drawOval( int x, int y, int width, int height ) ;

    public  abstract  void fillOval( int x, int y, int width, int height ) ;

    public  abstract  void drawArc( int x, int y, int width, int height,
                                    int startAngle, int arcAngle ) ;

    public  abstract  void fillArc( int x, int y, int width, int height,
                                    int startAngle, int arcAngle ) ;

        //  there are more polygon methods
    public  void drawPolygon( Polygon p ) ;

    public  void fillPolygon( Polygon p ) ;

    public  abstract  void drawString( String str, int x, int y ) ;

        //  there are more drawImage() methods
    public  abstract  boolean drawImage( Image img, int x, int y,
                                         ImageObserver observer ) ;
}
```

15.1 Graphics Coordinate System

The AWT coordinates are measured in pixels and the origin (0, 0) of the coordinate system is the top left corner of the frame. The standard coordinate system is shown below:



15.2 Simple Graphics Program

Output from simple drawing program

```
/*  SimpleDraw.java
 *
 *  Examples of drawing using the basic AWT graphics methods.
 *
 *  Bruce M. Bolden
 *  January 16, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.awt.*;

import corejava.*;

public class SimpleDrawFrame extends CloseableFrame
{
    static final int  FRAME_WIDTH  = 200;
    static final int  FRAME_HEIGHT = 200;
```

```
public static void main( String args[] )
{
  System.out.println( "Simple Draw" );

    SimpleDrawFrame f = new SimpleDrawFrame();
    f.setTitle( "Simple Draw" );
    f.show();
        //  must resize or the window is not visible!
    f.setSize(FRAME_WIDTH, FRAME_HEIGHT);

    try {
        System.in.read(); // prevent console window from going away
    } catch (java.io.IOException e) {}
}

public void paint( Graphics g )
{
    g.translate( getInsets().left, getInsets().top );

    g.drawLine( 0, 0, FRAME_WIDTH, FRAME_HEIGHT );
    g.setColor( Color.red );
    g.drawLine( 0, FRAME_HEIGHT, FRAME_WIDTH, 0 );

    g.drawString( "Center", FRAME_WIDTH/2, FRAME_HEIGHT/2 );

    g.drawRect( 20, 20, 20, 20 );

    g.setColor( Color.green );
    g.fillRect( 20, 50, 20, 20 );

    g.setColor( Color.blue );
    g.draw3DRect( 20, 80, 20, 20, true );

    g.setColor( Color.cyan );
    g.fill3DRect( 20, 110, 20, 20, true );

    g.drawOval( 50, 20, 20, 20 );
```

```
        g.setColor( Color.green );
        g.fillOval( 50, 50, 20, 20 );

        g.drawArc( 50, 80, 20, 20, 0, 180 );
        g.setColor( Color.blue );
        g.fillArc( 50, 110, 20, 20, 0, 90 );
    }
}
```

16 Programming Assignments

16.1 Program Development Model

1. Read specifications (assignment)
2. Outline/Define solution
   - Classes
   - Data structures, state variables, etc.
3. Review outline
   - Does your outline satisfy the specifications?
   - If not, repeat necessary steps above.
4. Begin implementation
5. Refine implementation
   - Does your implementation satisfy the specifications?
   - If not, repeat necessary steps above.
6. Testing
   - Design flaw (severity, time to fix?)
   - Operational errors (severity, time to fix?)
7. Finished Program
   - Documentation
   - Other notes

16.2 Other design/implementation issues

- flexibility
- maintainablity
- reusabilty

A good design recognizes the inevitability of change, and plans an accommodation for these activities from the very beginning.

16.3 Program Development Model Applied to the Checker Board Display Program

Consider the checker board assignment specifications.

16.3.1 Specifications

- Write a program that displays a checker board (use `fillRect` initially).
- Label the two opposing sides of the board.
- Label the *squares* of the checker board.
- In your display method, explore the use of colors and other shapes to form your checker board.

16.3.2 Display Checker Board

Considerations: size (window and square), location, color

16.3.3 Label Board Sides

Considerations: location, How to draw a label (string), size (font)

16.3.4 Label Squares

Considerations: size, location

16.3.5 Alternative Checker Board Displays

Considerations: shape, location

16.4 Program Development Model Applied to the LED Display Program

Consider the LED Digit Display assignment specifications.

16.4.1 Specifications

- Develop a class for displaying numbers using an LED segmented display format (example below).
- Explore the use of colors in displaying your digits.

16.4.2 LED Digit Class

Considerations: size, location, color (background and foreground)
Details: segment drawing (style, sizing)

LED segment numbering

17 Abstract Classes and Methods

Previously, we saw a more traditional approach to creating a class to handle different geometric shapes. During our discussion I pointed out that this approach is not particularly extensible because each time a new shape is added, the code must be modified to extend the class to handle the new shape. In addition, the programmer must know the *type code* of the object being created.

What is desired is some general purpose parent class that contains common information, e.g., position, color, etc. Java allows you to declare a class `abstract`, but the compiler will not allow you create an instance of it. Consider an abstract `GObject` class that will have methods to set and look up position, color, etc. (`getX()`, `setX()`, `setColor()`, etc.), but you cannot create a `GObject` object.

Note: The methods `get`*`XXX`* and `set`*`XXX`* are often called *accessors* and *mutators*.

17.1 Abstract Rules

Rules regarding `abstract` methods and the `abstract` classes that contain them[4]:

- Any class with an `abstract` method is automatically `abstract` itself, and must be declared as such.
- A class may be declared `abstract` even if it has no `abstract` methods. This prevents it from being instantiated.
- An `abstract` class cannot be instantiated.

[4] Java in a Nutshell, David Flanagan, O'Reilly, 1997

- A subclass of an `abstract` class can be instantiated if it overrides each of the `abstract` methods of its superclass and provides an implementation (i.e.., a method body) for all of them.
- If a subclass of an `abstract` class does not implement all of the `abstract` methods it inherits, that subclass is also `abstract`.

17.2 Sample Code

The following program calculates and displays the area of a circle, rectangle, and a square (well known geometric objects). Pay close attention to some of the comments that describe how to invoke methods in the testing code.
The basic output is shown below:

```
Area of circle:       314.159265358979
Area of circle2:      314.159265358979
Area of square:       100
Area of square2:      1225
Area of rect:       200
Area of rect2 :      1875
Color of circle:       java.awt.Color[r=0,g=0,b=0]
Color of circle2:      java.awt.Color[r=255,g=0,b=0]
Color of square:       java.awt.Color[r=0,g=255,b=0]
Color of square2:      java.awt.Color[r=0,g=0,b=255]
Color of rect:         java.awt.Color[r=0,g=255,b=0]
Color of rect2:        java.awt.Color[r=0,g=0,b=255]
Color of rect2:        java.awt.Color[r=255,g=255,b=255]
```

```
/*  GObject.java
 *
 *  Basic properties of geometric objects.
 *
 *  Bruce M. Bolden
 *  February 8, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.awt.Color;

/**  Parent class for all geometric objects.  */

public abstract class GObject
{
    protected int  x, y;
    protected Color  c;

    public int getX() {
       return  x;
    }

    public void setX(int x) {
       this.x = x;
    }

    public int getY() {
       return  y;
    }

    public void setY(int y) {
       this.y = y;
    }

    public Color getColor() {
       return  c;
    }
```

```
    public void setColor(Color c) {
       this.c = c;
    }
}
```

```
/*  Circle.java
 *
 *  Bruce M. Bolden
 *  February 8, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.awt.Color;
import java.awt.Graphics;

/**  Parent class for all circular objects.   */

final public class Circle extends GObject
{
    private int  radius;
    private static final int DEFAULT_RADIUS = 10;

    public Circle() {
        setX( 0 );    //  default position (0, 0)
        setY( 0 );
        setColor( Color.black );
        this.radius = DEFAULT_RADIUS;
    }

    public Circle(int x, int y, Color c) {
        this( x, y, c, DEFAULT_RADIUS );
    }

    public Circle(int x, int y, Color c, int r) {
        setX( x );
        setY( y );
        setColor( c );
        this.radius = r;
    }

    /**  calculates the area of a rectangular object  */
    public double area() {
        return  Math.PI * radius * radius;
```

```
    }

    /**  writes the area to standard output stream  <br>
     *   Note the use of the method area()  */
    public void showArea() {
        System.out.println( "     " + area() );
    }

    /**  writes the color to standard output stream  <br>
     *   Note the use of toString().  */
    public void showColor() {
        System.out.println( "     " + c.toString() );
    }

    /**  draw the circle  */
    public void draw( Graphics g ) {
        g.setColor( getColor() );
        g.drawOval( x, y, radius, radius );
    }
}
```

```
/*  Rectangle.java
 *
 *  Bruce M. Bolden
 *  February 8, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.awt.Color;
import java.awt.Graphics;

/**  Parent class for all rectangular objects.  */

public class Rectangle extends GObject
{
    private int  width;
    private int  height;
    private static final int DEFAULT_SIZE = 10;

    public Rectangle() {
        this( 0, 0, Color.black, DEFAULT_SIZE, DEFAULT_SIZE );
    }

    public Rectangle(int x, int y, Color c) {
        this( x, y, c, DEFAULT_SIZE, DEFAULT_SIZE );
    }

    public Rectangle(int x, int y, Color c, int width, int height) {
        setX( x );
        setY( y );
        setColor( c );
        this.width  = width;
        this.height = height;
    }

    /**  calculates the area of a rectangular object  */
    public int area() {
        return  width * height;
    }
```

```
    /**  writes the area to standard output stream  <br>
     *   Note the use of the method area()   */
    public void showArea() {
        System.out.println( "     " + area() );
    }

    /**  writes the color to standard output stream  <br>
     *   Note the use of toString().  */
    public void showColor() {
        System.out.println( "     " + c.toString() );
    }

    /**  draw the rectangle  */
    public void draw( Graphics g ) {
        g.setColor( getColor() );
        g.drawRect( x, y, width, height );
    }
}
```

```
/*  TestGObject.java
 *
 *  Bruce M. Bolden
 *  February 8, 1998
 *  http://www.cs.uidaho.edu/~bruceb/
 */

import java.io.*;
import java.awt.*;

public class TestGObject    extends  CloseableFrame
{
/*  Must declare objects as static
Error: Can't make a static reference to nonstatic variable circle in class
TestGObject. TestGObject.java line 31           circle.showArea();
*/
    static Circle circle  = new Circle();
    static Circle circle2 = new Circle( 50, 50, Color.red );
    static Square square  = new Square( 50, 50, Color.green );
    static Square square2 = new Square( 150, 150, Color.blue, 35 );
    static Rectangle rect  = new Rectangle( 100, 100, Color.green, 10, 20 );
    static Rectangle rect2 = new Rectangle( 200, 200, Color.blue, 25, 75 );

    public static void main( String[] args ) {
        TestGObject f = new TestGObject();
        f.resize( 400, 400 );
        f.show();
            //  Show the "properties" of the objects
        f.showObjectProperties();
    }

    /**  Draw the objects  */
    public void paint( Graphics g ) {
        circle.draw( g );
        circle2.draw( g );
        square.draw( g );
        square2.draw( g );
        rect2.draw( g );
```

```
    }

    /**  Show the "properties" of the objects  */
    private void showObjectProperties() {
        System.out.print( "Area of circle:  " );
        TestGObject.circle.showArea();
        System.out.print( "Area of circle2: " );
        circle2.showArea();
        System.out.print( "Area of square:  " );
        square.showArea();
        System.out.print( "Area of square2: " );
        square2.showArea();
        System.out.print( "Area of rect:  " );
        rect.showArea();
        System.out.print( "Area of rect2 : " );
        rect2.showArea();

        System.out.print( "Color of circle:  " );
        circle.showColor();
        System.out.print( "Color of circle2: " );
        circle2.showColor();
        System.out.print( "Color of square:  " );
        square.showColor();
        System.out.print( "Color of square2: " );
        square2.showColor();
        System.out.print( "Color of rect:    " );
        rect.showColor();
        System.out.print( "Color of rect2:   " );
        rect2.showColor();

         //  change color
        rect2.setColor( Color.white );
        System.out.print( "Color of rect2:   " );
        rect2.showColor();
    }
}
```

18 Interfaces

It is frequently desirable for classes to implement specific methods that perform some function in addition to some other operations. Without multiple inheritance, this is difficult, if not impossible to do. The solution to this problem in Java is *interfaces*. An interface looks a lot like an abstract class, except that it uses `interface` instead of `abstract class`.

Recall that an `abstract` class may define some `abstract` methods and some non-`abstract` methods, all the methods defined within an interface are implicitly `abstract`. All variables declared in an interface must be `static` and `final`— constants.

So what good are `interfaces`? We have used them a considerable amount for all of the event-handling programs we have written. Just as a class `extends` its superclass, it can also `implement` an interface. `implements` is a keyword that can appear in a class declaration following the `extends` clause. `implements` must be followed by the name of the interface the class implements.

To implement an interface, a class must first declare the interface in an `implements` clause, and then it must provide an implementation (the code) for all of the methods described in the interface.

18.1 ObjGeometry.java

```
1 /*   ObjGeometry.java
2  */
3
4 interface ObjGeometry
5 {
6     double area();
7
```

```
8     double circumference();
9 }
```

18.2 Circle.java

```
1 /*  Circle.java
2  *
3  *  Bruce M. Bolden
4  *  April 20, 1998
5  *  http://www.cs.uidaho.edu/~bruceb/
6  */
7
8 import java.awt.Color;
9 import java.awt.Graphics;
10
11 /**  Parent class for all circular objects.   */
12
13 final public class Circle
14     extends GObject
15     implements ObjGeometry
16 {
17     private int  radius;
18     private static final int DEFAULT_RADIUS = 10;
19
20     public Circle() {
21         setX( 0 );    //  default position (0, 0)
22         setY( 0 );
23         setColor( Color.black );
24         this.radius = DEFAULT_RADIUS;
25     }
26
27     public Circle(int x, int y, Color c) {
28         this( x, y, c, DEFAULT_RADIUS );
29     }
30
31     public Circle(int x, int y, Color c, int r) {
32         setX( x );
33         setY( y );
34         setColor( c );
35         this.radius = r;
36     }
37
38     /**  calculates the area of a circular object  */
39     public double area() {
40         return  Math.PI * radius * radius;
```

```
41      }
42
43      /**  writes the area to standard output stream  <br>
44       *   Note the use of the method area()  */
45      public void showArea() {
46          System.out.println( "␣␣␣␣␣" + area() );
47      }
48
49      /**  calculates the circumference of a circular object  */
50      public double circumference() {
51          return  2.0 * Math.PI * radius;
52      }
53
54      /**  writes the color to standard output stream  <br>
55       *   Note the use of toString().  */
56      public void showColor() {
57          System.out.println( "␣␣␣␣␣" + c.toString() );
58      }
59
60      /**  writes the circumference to standard output stream  <br>
61       *   Note the use of the method area()   */
62      public void showCircumference() {
63          System.out.println( "␣␣␣␣␣" + area() );
64      }
65
66      /**  draw the circle  */
67      public void draw( Graphics g ) {
68          g.setColor( getColor() );
69          g.drawOval( x, y, radius, radius );
70      }
71 }
72
```

18.3 TestGObject.java

```
 1 /*  TestGObject.java
 2  *
 3  *  Bruce M. Bolden
 4  *  April 20, 1998
 5  *  http://www.cs.uidaho.edu/~bruceb/
 6  */
 7
 8 import java.io.*;
 9 import java.awt.*;
10
```

```
11 public class TestGObject extends Frame
12 {
13 /* Must declare objects as static
14 Error: Can't make a static reference to nonstatic variable circle in class Te
15 */
16     static Circle circle  = new Circle();
17     static Circle circle2 = new Circle( 50, 50, Color.red );
18     static Square square  = new Square( 50, 50, Color.green );
19     static Square square2 = new Square( 150, 150, Color.blue, 35 );
20     static Rectangle rect  = new Rectangle( 100, 100, Color.green, 10, 20 );
21     static Rectangle rect2 = new Rectangle( 200, 200, Color.blue, 25, 75 );
22
23     public static void main( String[] args ) {
24         TestGObject f = new TestGObject();
25         f.resize( 400, 400 );
26         f.show();
27
28         // Show the "properties" of the objects
29         f.showObjectProperties();
30     }
31
32     /** Draw the objects */
33     public void paint( Graphics g ) {
34         circle.draw( g );
35         circle2.draw( g );
36         square.draw( g );
37         square2.draw( g );
38         rect2.draw( g );
39     }
40
41     /** Show the "properties" of the objects */
42     private void showObjectProperties() {
43         System.out.print( "Area of circle:  " );
44         TestGObject.circle.showArea();
45         System.out.print( "Area of circle2: " );
46         circle2.showArea();
47
48         System.out.print( "Circumference of circle:  " );
49         TestGObject.circle.showCircumference();
50         System.out.print( "Circumference of circle2: " );
51         circle2.showCircumference();
52         System.out.print( "Circumference of square:  " );
53         square.showCircumference();
54         System.out.print( "Circumference of square2: " );
55         square2.showCircumference();
```

```
56            System.out.print( "Circumference of rect:  " );
57            rect.showCircumference();
58            System.out.print( "Circumference of rect2 : " );
59            rect2.showCircumference();
60        }
61 }
```

18.4 Implementing Multiple Interfaces

When a class implements more than one interface, it means that if must provide an implementation for all of the abstract methods in all of its interfaces.

A class that implements multiple interfaces uses a list of comma-separated interfaces in the `implements` clause of its definition.

```
public class  Rectangle   extends GObject
    implements  ObjGeometry, ObjProperties
{
    ....
}
```

18.5 Constants in Interfaces

As previously noted, constants may appear in interface definitions. What does it mean to implement an interface that contains constants? It means that the class that implements the interface *inherits* the constants and can use them as if they were defined directly in the class.

Note: Constants do not need a prefix with the name of the interface and no implementation is required.

```
class One  {  static final double  k = 4.59; }
```

```
interface IntOne { static final double PI = 3.14159; }

class Two  implements IntOne
{
    void m( double r )
    {
        double x = One.k;
        double a = PI * r * r;
    }
}
```

18.6 Extending Interfaces

Interfaces can have sub-interfaces, just like classes can have sub-classes. A sub-interface inherits all the abstract methods and constants of its super-interface, and may define new abstract methods and constants. Interfaces are different from classes in one very important way.

An interface can extend more than one interface at a time.

```
public interface Transformable
    extends Scalable, Rotatable {}
public interface DrawableObject
    extends Drawable, Transformable {}
public class GShape
    implements DrawableObject { ... }
```

An interface that extends more than one interface inherits all the abstract methods and constants from each of those interfaces. It may also define its own additional abstract methods and constants. A class that implements such an interface must

implement the abstract methods defined in the interface and all the abstract methods inherited from all of the super-interfaces.

19 Vectors

One frequently used data structure is a vector. In many programming languages, vectors are implemented as arrays (fixed size, hold items of some homogeneous type). Java provides a `Vector` class that allows the size to change at run time and can hold heterogeneous objects. It is easiest to think of vectors as array-like objects that can expand and contract automatically without having to write code to manage these operations.

There are several important differences between Java vectors and Java arrays:

- Arrays in Java can hold any type, including numbers and all class types.
- Vectors in Java must hold instances of `Object`.
- The size of a vector is determined by the `size()` method.

The `Vector` class is not a part of the Java language. It is a utility class programmed by someone and supplied in the standard library.

19.1 Vector API

```
1 public class java.util.Vector
2     extends java.lang.Object
3     implements java.lang.Cloneable , java.io.Serializable
4 {
5     protected java.lang.Object elementData[];
6     protected int elementCount;
7     protected int capacityIncrement;
8
9     public java.util.Vector(int, int);
10    public java.util.Vector(int);
11    public java.util.Vector();
```

```
12
13     public final synchronized void copyInto(java.lang.Object[]);
14     public final synchronized void trimToSize();
15     public final synchronized void ensureCapacity(int);
16     public final synchronized void setSize(int);
17
18     public final int capacity();
19     public final int size();
20     public final boolean isEmpty();
21
22     public final synchronized java.util.Enumeration elements();
23     public final boolean contains(java.lang.Object);
24     public final int indexOf(java.lang.Object);
25     public final synchronized int indexOf(java.lang.Object, int);
26     public final int lastIndexOf(java.lang.Object);
27     public final synchronized int lastIndexOf(java.lang.Object, int);
28     public final synchronized java.lang.Object elementAt(int);
29     public final synchronized java.lang.Object firstElement();
30     public final synchronized java.lang.Object lastElement();
31     public final synchronized void setElementAt(java.lang.Object, int);
32     public final synchronized void removeElementAt(int);
33     public final synchronized void insertElementAt(java.lang.Object, int);
34     public final synchronized void addElement(java.lang.Object);
35     public final synchronized boolean removeElement(java.lang.Object);
36     public final synchronized void removeAllElements();
37     public synchronized java.lang.Object clone();
38     public final synchronized java.lang.String toString();
39 }
```

19.2 Manipulating Vectors

19.2.1 Creating a Vector

A new vector is created by specifying its initial *capacity* in the `Vector` constructor.

```
Vector  objVector = new Vector(3);
```

There is an important distinction between the capacity of a vector and the size of an array. If an array is allocated with three entries, then the array has three *slots*. A vector with a capacity of three elements has the potential of holding three

elements (actually more), but at the beginning, a vector holds no elements.

19.2.2 Adding an element to a Vector

To add an element to a vector, use the `addElement()` method.

```
Circle circle = new Circle();
Square square = new Square();

objVector.addElement( circle );
objVector.addElement( square );
```

19.2.3 Vector Sizing Operations

The `size()` method returns the number of elements in a vector. So,

```
objVector.size();
```

is the equivalent of

```
objArray.length;
```

for an array. The size of a vector is always less than or equal to its capacity.

When you are reasonably sure that the vector is at its permanent size, you can use the `trimToSize()` method. This method adjusts the size of the memory block to use exactly as much storage as is required to hold the current number of elements. The garbage collector will reclaim any excess memory.

Note: Once a vector's size has been trimmed, adding new elements will move the block again, which takes time. `trimToSize()`

should only be used when you are sure you won't add any more elements to the vector.

19.2.4 Accessing Vector Elements

1. Accessing vector elements is not as simple as accessing array elements (no `a[i]`).

Array	**Vector**
`x = a[i];`	`x = v.elementAt(i);`
`a[i] = x;`	`v.setElementAt(x, i);`

2. The `Vector` class holds elements of any type—a sequence of `Object`s.

Note: Vectors, like arrays, are zero-based.

On rare occasions, vectors are useful for storing *heterogeneous collections*. Objects of completely unrelated classes are inserted into a vector on purpose. When retrieving a vector entry, the type of every retrieved object must be tested. This is illustrated in the following code fragment:

```
Vector gObjVector = new Vector(3);

    //  create and store objects
Circle circle2 = new Circle( 30, 30, Color.red, 20 );
Square square2 = new Square( 70, 70, Color.blue, 35 );

gObjVector.addElement( circle2 );
gObjVector.addElement( square2 );

...

    //  retrieve and display objects
Object obj = gObjVector.elementAt(i);

if( obj instanceof Circle )
{
    Circle circle = (Circle)obj;
    circle.draw(g);
}
else if( obj instanceof Square )
{
    Square square = (Square)obj;
    square.draw(g);
}
```

Note: This is not a particularly good way to write code. Why?

19.2.5 Inserting/Removing Elements in the Middle of a Vector

Elements can also be inserted in the middle of a vector.

```
gObjVector.insertElementAt(circle, n);
```

The elements at location `n` and above are shifted up to make room for the new entry. After the insertion, if the new size of the vector after the insertion exceeds the vector's capacity, Java reallocates its storage.

Similarly, an element can be removed from the middle of a vector.

```
Object obj = gObjVector.removeElementAt(n);
```

The elements located above it are copied down, and the size is reduced by one.

19.3 Example using the GObject and Vector Classes

Previously, an example that illustrated the concept of `abstract` classes was presented. One drawback of that solution was that it required the objects to be displayed to be visible to the entire class. A second, and more important, drawback is that the number of objects could not be increased easily (code to create and to display the object had to be added).

The following solution, is a bit more robust, but has its own set of problems as we shall see.

Output from modified GObject program

```
1 /*   TestGObjectVector.java
2  *
3  *   Bruce M. Bolden
4  *   March 8, 1998
5  *   http://www.cs.uidaho.edu/~bruceb/
6  */
7
8 import java.io.*;
9 import java.awt.*;
10 import java.util.Vector;
11
12 public class TestGObjectVector    extends  Frame
13 {
14     private static Vector gObjVector = new Vector(3);
15
16     public static void main( String[] args ) {
17         TestGObjectVector f = new TestGObjectVector();
18         f.resize( 200, 200 );
19         f.show();
20
21         InitObjectVector();
22
23         System.out.println( "\nObject␣properties␣(in␣main():" );
24         f.showObjectProperties();
25     }
26
27     /** Draw the objects  */
28     public void paint( Graphics g ) {
29         drawObjects( g );
30
31         System.out.println( "\nObject␣properties␣(in␣paint()):" );
32         showObjectProperties();
33     }
34
35     static private void drawObjects( Graphics g )
36     {
37         for( int i = 0 ; i < gObjVector.size() ; ++i )
38         {
39             Object obj = gObjVector.elementAt(i);
40
41             if( obj instanceof Circle )
42             {
43                 Circle circle = (Circle)obj;
44                 circle.draw(g);
45             }
```

```
46             else if( obj instanceof Square )
47             {
48                 Square square = (Square)obj;
49                 square.draw(g);
50             }
51             else if( obj instanceof Rectangle )
52             {
53                 Rectangle rect = (Rectangle)obj;
54                 rect.draw(g);
55             }
56         }
57     }
58 
59     static private void InitObjectVector()
60     {
61         Circle circle   = new Circle();
62         Circle circle2 = new Circle( 30, 30, Color.red, 20 );
63         Square square  = new Square( 50, 50, Color.green );
64         Square square2 = new Square( 70, 70, Color.blue, 35 );
65         Rectangle rect  = new Rectangle(  90, 90, Color.green, 10, 20 );
66         Rectangle rect2 = new Rectangle( 100, 40, Color.blue, 25, 75 );
67 
68         System.out.println( "InitObjectVector:" );
69         System.out.println( "\tgObjVector.size(): " +
70                             gObjVector.size() );
71 
72             // add objects to our vector
73         gObjVector.addElement( circle  );
74         gObjVector.addElement( circle2 );
75         gObjVector.addElement( square  );
76         gObjVector.addElement( square2 );
77         gObjVector.addElement( rect    );
78         gObjVector.addElement( rect2   );
79 
80         System.out.println( "\tgObjVector.size(): " +
81                             gObjVector.size() );
82 
83         gObjVector.trimToSize();
84     }
85 
86 
87     /**  Show the "properties" of the objects  */
88     private void showObjectProperties()
89     {
90         for( int i = 0 ; i < gObjVector.size() ; ++i )
```

```
 91            {
 92                  // retrieve an object from the vector
 93                  Object obj = gObjVector.elementAt(i);
 94
 95                  if( obj instanceof Circle )
 96                  {
 97                       Circle circle = (Circle)obj;
 98                       System.out.print( "Area of circle:  " );
 99                       circle.showArea();
100                       System.out.print( "Color of circle: " );
101                       circle.showColor();
102
103                       circle.setColor( Color.red );
104                  }
105                  else if( obj instanceof Square )
106                  {
107                       Square square = (Square)obj;
108                       System.out.print( "Area of square:  " );
109                       square.showArea();
110                       System.out.print( "Color of square: " );
111                       square.showColor();
112
113                       square.setColor( Color.green );
114                  }
115                  else if( obj instanceof Rectangle )
116                  {
117                       Rectangle rect = (Rectangle)obj;
118                       System.out.print( "Area of rect:    " );
119                       rect.showArea();
120                       System.out.print( "Color of rect:   " );
121                       rect.showColor();
122
123                       rect.setColor( Color.blue );
124                  }
125            }
126      }
127 }
```

19.3.1 Output written to `System.output`

```
InitObjectVector:
    gObjVector.size(): 0
    gObjVector.size(): 6

Object properties (in main():
Area of circle:       314.159265358979
Color of circle:      java.awt.Color[r=0,g=0,b=0]
Area of circle:       1256.637061435916
Color of circle:      java.awt.Color[r=255,g=0,b=0]
Area of square:       100
Color of square:      java.awt.Color[r=0,g=255,b=0]
Area of square:       1225
Color of square:      java.awt.Color[r=0,g=0,b=255]
Area of rect:         200
Color of rect:        java.awt.Color[r=0,g=255,b=0]
Area of rect:         1875
Color of rect:        java.awt.Color[r=0,g=0,b=255]

Object properties (in paint()):
Area of circle:       314.159265358979
Color of circle:      java.awt.Color[r=255,g=0,b=0]
Area of circle:       1256.637061435916
Color of circle:      java.awt.Color[r=255,g=0,b=0]
Area of square:       100
Color of square:      java.awt.Color[r=0,g=255,b=0]
Area of square:       1225
Color of square:      java.awt.Color[r=0,g=255,b=0]
Area of rect:         200
Color of rect:        java.awt.Color[r=0,g=0,b=255]
```

```
Area of rect:          1875
Color of rect:         java.awt.Color[r=0,g=0,b=255]
```

20 Reflection

The original implementation of `Class` in Java was fairly limited. In Java 1.1, many methods were added to the implementation of `Class` to enable programmers to write programs that manipulate Java code dynamically. The primary reason for this enhancement is *Java Beans*, the component architecture for Java.[5]

A program that can analyze the capabilites of classes is called *reflective*. The reflection methods can be used to:

- Analyze the capabilities of classes at run time.
- Inspect objects at runtime.
- Implement generic array manipulation code.

20.1 Using Reflection to Analyze the Capabilities of Classes

There are three classes in `java.lang.reflect`: `Field`, `Method`, and `Constructor`. These describe the data fields, the operations, and the constructors of a class. All three classes have a method called `getName` that returns the name of the item.

The `Field` class has a method `getType` that returns an object (of type `Class`), that describes the field type.

The `Method` and `Constructor` classes have methods to report the return type and types of the parameters used for the methods.

The `getFields`, `getMethods()`, and `getConstructor()` methods of `Class` return arrays of the *public* fields, operations, and constructors that the class supports.

[5]Material derived from, Core Java, Volume I, Horstmann and Cornell, 1997

The methods `getDeclaredFields`, `getDeclaredMethods()`, and `getDeclaredConstructor()` of `Class` return arrays of all fields, operations, and constructors that the class supports.

20.2 Example: Analyzing Classes using Reflection

```
1 /*   ReflectionTest.java
2  *
3  *   An example of the reflection properties of Java.   This
4  *   example is derived from Example 5-4 in Core Java, Volume I.
5  *   Performs the same basic functions as javap (seems faster!).
6  *
7  *   Bruce M. Bolden
8  *   March 10, 1998
9  *   http://www.cs.uidaho.edu/~bruceb/
10 *
11 */
12
13 import java.lang.reflect.*;
14
15 public class ReflectionTest
16 {
17     public static void main(String[] args)
18     {
19         int nArgs = args.length;
20         if( nArgs < 1 )
21         {
22             ShowUsage();
23             System.exit(-1);
24         }
25
26             // Process all classes
27         for( int i = 0 ; i < nArgs ; ++i )
28         {
29             if( i > 0 )
30                 System.out.println( "\n\n" );
31
32             String name = args[i];
33             processClass( name );
34         }
35     }
36
37     /** Process a single class name */
38     public static void processClass( String name )
39     {
40         try
41         {
42             Class cl = Class.forName(name);
43             Class sc = cl.getSuperclass();
```

```
44
45            System.out.print("class␣" + name);
46            if (sc != null && !sc.equals(Object.class))
47                System.out.print("\n␣␣␣␣extends␣" + sc.getName());
48            if( sc != null && !sc.equals(Object.class) )
49            {
50                System.out.print("\n␣␣␣␣implements␣" );
51                printInterfaces(cl);
52            }
53            System.out.print("\n{\n");
54
55            System.out.println( "\t//␣␣Constructors" );
56            printConstructors(cl);
57            System.out.println();
58
59            System.out.println( "\t//␣␣Methods" );
60            printMethods(cl);
61            System.out.println();
62
63            System.out.println( "\t//␣␣Fields" );
64            printFields(cl);
65
66            System.out.println("}");
67        }
68        catch(ClassNotFoundException e)
69        {
70            System.out.println("Class␣not␣found.");
71        }
72    }
73
74    /** Show Interfaces implemented by the class */
75    public static void printInterfaces(Class cl)
76    {
77        Class[] impClasses = cl.getInterfaces();
78
79        for (int i = 0; i < impClasses.length; i++)
80        {
81            Class c = impClasses[i];
82            String name = c.getName();
83
84            if( i > 0 )
85                 System.out.print(",␣␣" + name );
86            else
87                 System.out.print("␣" + name );
88        }
```

```
89      }
90
91      /**  Show Constructors implemented by the class  */
92      public static void printConstructors(Class cl)
93      {
94         Constructor[] constructors = cl.getDeclaredConstructors();
95
96         for (int i = 0; i < constructors.length; i++)
97         {
98           Constructor c = constructors[i];
99            Class[] paramTypes = c.getParameterTypes();
100           String name = cl.getName();
101
102           System.out.print(Modifier.toString(c.getModifiers()));
103           System.out.print(" " + name + "(");
104           for (int j = 0; j < paramTypes.length; j++)
105           {
106              if (j > 0) System.out.print(", ");
107              System.out.print(paramTypes[j].getName());
108           }
109           System.out.println(");");
110        }
111     }
112
113     /**  Show Methods implemented by the class  */
114     public static void printMethods(Class cl)
115     {
116        Method[] methods = cl.getDeclaredMethods();
117
118        for (int i = 0; i < methods.length; i++)
119        {
120           Method m = methods[i];
121           Class retType = m.getReturnType();
122           Class[] paramTypes = m.getParameterTypes();
123           String name = m.getName();
124
125           System.out.print(Modifier.toString(m.getModifiers()));
126           System.out.print(" " + retType.getName() + " " + name + "(");
127
128           for (int j = 0; j < paramTypes.length; j++)
129           {
130              if (j > 0) System.out.print(", ");
131              System.out.print(paramTypes[j].getName());
132           }
133           System.out.println(");");
```

```
134         }
135     }
136
137     /**  Show Fields implemented by the class  */
138     public static void printFields(Class cl)
139     {
140         Field[] fields = cl.getDeclaredFields();
141
142         for (int i = 0; i < fields.length; i++)
143         {
144             Field f = fields[i];
145             Class type = f.getType();
146             String name = f.getName();
147             System.out.print(Modifier.toString(f.getModifiers()));
148             System.out.println(" " + type.getName() + " " + name + ";");
149         }
150     }
151
152     /**  Show expected argument list  */
153     static void ShowUsage()
154     {
155            System.out.println( "Usage:  java ReflectionTest className[s]" );
156     }
157 }
```

21 Events

Event Handling is important for programs with a graphical user interface (GUI). It is very important to understand how Java handles events to create truly useful and *robust* programs.[6]

21.1 Event Handling Basics

General Event Handling Process

[6]Material derived from Chapter 8 of Core Java, Volume 1, Horstmann and Cornell, 1997

How event handling in the JDK 1.1 works:

- A listener object is an instance of a class that implements a special interface called a *listener interface*.
- An event source is an object that can register listener objects and send them notifications when events occur. These notifications are methods of the listener interface.

The listener object is registered with the source object with code of the following form:

```
eventSourceObject.addEventListener(eventListenerObject);
```

For example,

```
Button  b = new Button("Clear");}
b.addActionListener(canvas);
```

The `canvas` object is notified whenever an "action event" occurs in the button.

To implement the `ActionListener` interface, the listener class must have a method (called `actionPerformed`) that receives an `ActionEvent` object as a parameter.

There are eleven listener interfaces in the `java.awt.Event` package:

ActionListener	KeyListener
AdjustmentListener	MouseListener
ComponentListener	MouseMotionListener
ContainerListener	TextListener
FocusListener	WindowListener
ItemListener	

Clearly, a lot of classes and interfaces. The principle is fairly simple: *A class that is interested in receiving events registers itself with the event source. It then gets the events that it asked for during registration.*

21.1.1 Capturing Window Events

Implementing a closable frame class.

`class CloseableFrame implements WindowListener`

By doing this, `CloseableFrame` objects can listen to window events.

```
public class CloseableFrame implements WindowListener
{
   public CloseableFrame()
   {
      addWindowListener(new WindowAdapter(this);
      show();
   }
}
```

There are seven methods in the WindowListener interface.

```
void windowClosed( WindowEvent e )
void windowIconified( WindowEvent e )
void windowOpened( WindowEvent e )
void windowClosing( WindowEvent e )
void windowDeiconified( WindowEvent e )
void windowActivated( WindowEvent e )
void windowDeactivated( WindowEvent e )
```

Any class that implements this interface must implement all seven of its methods! This is illustrated in the following example.

```
public class CloseableFrame
   extends Frame
        implements WindowListener
{
   public CloseableFrame()
   {
      setTitle( "Closeable Frame" );
      setSize( 300, 300 );
      addWindowListener(this);
   }

   void windowClosed( WindowEvent e ) {System.exit(0);}
   void windowIconified( WindowEvent e ) {}
   void windowOpened( WindowEvent e ) {}
   void windowClosing( WindowEvent e ) {}
   void windowDeiconified( WindowEvent e ) {}
   void windowActivated( WindowEvent e ) {}
   void windowDeactivated( WindowEvent e ) {}

   public static void main( String[] args )
   {
       CloseableFrame f = new CloseableFrame();
       f.show()
   }
}
```

21.1.2 Adapter Classes

Each listener interface has a companion *adapter* class that implements all the methods in the interface but does nothing with them. For example, the `WindowAdapter` class has seven *do-nothing* methods.

The AWT has an adapter class for each of the seven listener interfaces that has more than one method.

Adapter classes:

ComponentAdapter	MouseAdapter
ContainerAdapter	MouseMotionAdapter
FocusAdapter	WindowAdapter
KeyAdapter	

We only need to override one of the seven methods.

```
class WindowCloser extends WindowAdapter
{
    void windowClosing(WindowEvent e)
    {
       System.exit(0);
    }
}
```

Using this new class, we can register an object of type `WindowCloser` as the event listener, so that the frame class does not need to implement the listener interface.

```
public class CloseableFrame extends Frame
{
    public CloseableFrame()
    {
```

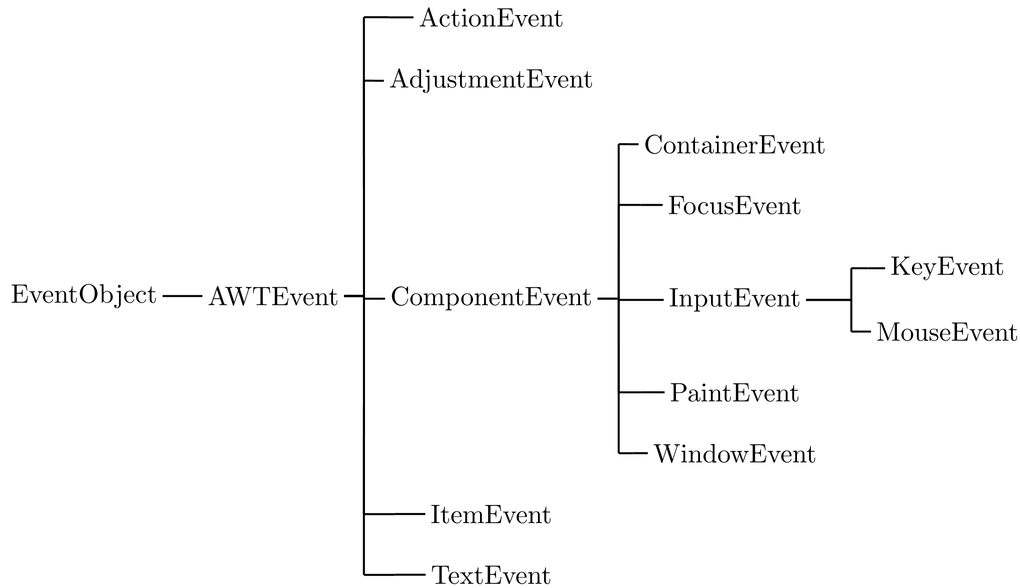
```
        WindowCloser wc = new WindowCloser();
        addWindowListener(wc);
        ...
    }
}
```

This is simpler if the listener class is an *anonymous inner class*.

```
public class CloseableFrame extends Frame
{
   public CloseableFrame()
   {
      addWindowListener( new WindowAdapter() {
         public void  windowClosing(WindowEvent e)
         { System.exit(0); } }
      );
      setSize(300, 200);
      setTitle(getClass().getName());
   }
}
```

21.2 JDK 1.1 Event Hierarchy

Inheritance Diagram of the AWT Event Classes



When a source object needs to tell a listener object that an event happened, the AWT:

- Calls the appropriate method of the listener interface
- Passes it an object that descends from `EventObject`.

Event objects encapsulate events. This happens automatically when the listener is registered with the source.

AWT event types that are passed to listeners:

ActionEvent	FocusEvent	MouseEvent
AdjustmentEvent	ItemEvent	TextEvent
ComponentEvent	KeyEvent	WindowEvent
ContainerEvent		

21.2.1 Example: Button Selection

```
/*  ButtonTest.java
 *
 *  From CoreJava I, Horstmann and Cornell
 *
 *  March 2, 1998
 */

import java.awt.*;
import java.awt.event.*;
import corejava.*;

public class ButtonTest extends CloseableFrame
    implements ActionListener
{
    public static void main( String[] args )
    {
        ButtonTest f = new ButtonTest();
        f.show();
    }

    public ButtonTest()
    {
        setLayout( new FlowLayout() );

        Button redButton = new Button( "Red" );
        add( redButton );
        redButton.addActionListener( this );

        Button orangeButton = new Button( "Orange" );
```

```
        add( orangeButton );
        orangeButton.addActionListener( this );

        Button yellowButton = new Button( "Yellow" );
        add( yellowButton );
        yellowButton.addActionListener( this );

        Button greenButton = new Button( "Green" );
        add( greenButton );
        greenButton.addActionListener( this );

        Button blueButton = new Button( "Blue" );
        add( blueButton );
        blueButton.addActionListener( this );
    }

    public void actionPerformed( ActionEvent e )
    {
        String arg = e.getActionCommand();
        Color c = Color.black;
        if(      arg.equals("Red"   ) ) c = Color.red;
        else if( arg.equals("Orange") ) c = Color.orange;
        else if( arg.equals("Yellow") ) c = Color.yellow;
        else if( arg.equals("Green" ) ) c = Color.green;
        else if( arg.equals("Blue"  ) ) c = Color.blue;
        setBackground( c );
        repaint();
    }
}
```

Sample Output from ButtonTest

22 Events

Event Handling is important for programs with a graphical user interface (GUI). It is very important to understand how Java handles events to create truly useful and *robust* programs.[7]

22.1 Semantic and Low-Level Events

The AWT makes a distinction between *low-level* and *semantic* events. A semantic event is one that expresses what the user is doing, such as "click that button." Low-level events are those events that make this possible.

There are four semantic event classes in the `java.awt.event` package:

- `ActionEvent` (button click, menu selection, double click on a list box item, clicking [Enter] in a text field)
- `AdjustmentEvent` (the user adjusted a scroll bar)
- `ItemEvent` (the user made a selection from a set of check-box or list items)
- `TextEvent` (the contents of a text box or text area were changed)

There are six low-level event classes:

- `ComponentEvent` (the component was resized, moved, shown, or hidden); the base class for all low-level events
- `KeyEvent` (a key was pressed or released)

[7]Material derived from Chapter 8 of Core Java, Volume 1, Horstmann and Cornell, 1997

- `MouseEvent` (the mouse button was depressed, released, moved of dragged)
- `FocusEvent` (a component got focus, lost focus)
- `WindowEvent` (the window was activated, deactivated, iconified, deiconified, or closed)
- `ContainerEvent` (notifies you that a component has been added or removed)

22.1.1 Event Handling Summary

Interface	Methods	Parameter	Events generated by
ActionListener	actionPerformed	ActionEvent getActionCommand getModifiers	Button List MenuItem TextField
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent getAdjustable getAdjustmentType getValue	Scrollbar
ItemListener	itemStateChanged	ItemEvent getItem getItemSelectable getStateChange	Checkbox CheckboxMenuItem Choice List
TextListener	textValueChanged	TextEvent	TextComponent
ComponentListener	componentMoved componentHidden componentResized	ComponentEvent getComponent	Component
ContainerListener	componentAdded componentRemoved	ContainerEvent getChild getContainer	Container
FocusListener	focusGained focusLost	FocusEvent IsTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent getKeyChar getKeyCode getKeyModifiersText getKeyText IsActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent getClickCount getX getY getPoint TranslatePoint IsPopupTrigger	Component
MouseMotionListener	mouseDragged mouseMoved		Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent getWindow	Window

22.1.2 Event Delegation Mechanism Review

Event *sources* are user interface components, windows, and menus. The operating system notifies an event source about interesting activities (e.g., mouse moves or keystrokes).

The event source describes the nature of the event in an *event object*. It also keeps a list of *listeners*, objects that want to be called when the event happens. The event source then calls the appropriate method of the *listener interface* to deliver information about the event to the various listeners. The source does this by passing the appropriate event object to the method in the listener class. The listener analyzes the event object to find out more about the event. For example, the `getX()` and `getY()` methods of the `MouseEvent` class tell the listener the current location of the mouse.

With one exception, each event type corresponds to a listener interface. The one exception is that both `MouseListener` and `MouseMotionListener` receive `MouseEvent` objects. This is done for efficiency—there are a lot of mouse events as the user moves the mouse around, and a listener that just cares about mouse *clicks* will not be bothered with unwanted mouse *moves*.

Java supplies a corresponding *adapter* class to all listener interfaces with more than one method. The adapter class defines all the methods of the interface to do nothing. Use adapter classes as a time- saving tool: use them when you want to override just a few of the listener interface methods.

The high-level semantic events are natural for GUI programming—they correspond to user input. To better understand the low level events, let us review some terminology.

- A *component* is a user interface element such as a button, text field, or scroll bar.
- A *container* is a screen area or component that can contain components, such as a window or an applet.

Recall that it is the job of the layout manager to arrange components inside a container. We will see more about this later.

All low-level events inherit from `ComponentEvent`. This class has a method called `getComponent` which reports the component that originated the event. For example, if a key event was fired because of an input into a text field, then `getComponent` returns a reference to that text field.

Note that the `PaintEvent` is not intended for programming—it is for the convenience of the AWT implementors.

23 Event Handling for Java Applications

23.1 Example: Java

```
1 /*   MenuTest.java
2  *
3  *   Bruce M. Bolden
4  *   March 5, 1998
5  */
6
7 import java.awt.*;
8 import java.awt.event.*;
9 import java.util.Random;
10
11 import corejava.*;
12
13
14 public class MenuTest extends CloseableFrame
15     implements ActionListener
16 {
17     protected Color textColor = Color.magenta;
18
19     public static void main(String args[]) {
20         MenuTest f = new MenuTest();
21
22         InitializeMenu( f );
23
24         f.setSize( 300, 300 );
25         f.show();
26     }
27
28     /**  Event handler
29      */
30     public void actionPerformed( ActionEvent e )
31     {
32         Color oldColor = textColor;
33         String cmd = e.getActionCommand();
34
35         if( cmd.equals("clear") )          //  File menu
36             clear();
37         else if( cmd.equals("quit") )
38             System.exit(0);
39         else if( cmd.equals("undo") )      //  Edit menu
40             /* do nothing  */ ;
41         else if( cmd.equals("cut") )
```

```
42                /* do nothing */ ;
43            else if( cmd.equals("copy") )
44                /* do nothing */ ;
45            else if( cmd.equals("paste") )
46                /* do nothing */ ;
47            else if( cmd.equals("red") )        // Color menu
48                textColor = Color.red ;
49            else if( cmd.equals("green") )
50                textColor = Color.green ;
51            else if( cmd.equals("blue") )
52                textColor = Color.blue ;
53
54            if( oldColor != textColor )
55                repaint();    // Note: calls paint() indirectly!
56        }
57
58        /** Clear the graphics context.
59         */
60        protected void clear()
61        {
62            Graphics g = this.getGraphics();
63            g.setColor( this.getBackground() );
64            g.fillRect( 0, 0, this.getSize().width, this.getSize().height );
65        }
66
67        public void paint( Graphics g )
68        {
69            Random r = new Random();
70
71            for( int i = 0 ; i < 100 ; i++ )
72            {
73                int x = r.nextInt() % this.getSize().width;
74                int y = r.nextInt() % this.getSize().height;
75
76                g.setColor( textColor );
77                g.drawString( "Hello!", x, y );
78            }
79        }
80
81        /** Initialize the menu
82         */
83        protected static void InitializeMenu( MenuTest f )
84        {
85            MenuBar mBar = new MenuBar();
86            f.setMenuBar( mBar );
```

```
87
88            //  Create/Add menus to Menu bar
89        Menu file   = new Menu( "File"   );
90        Menu edit   = new Menu( "Edit"   );
91        Menu color  = new Menu( "Color" );
92
93        mBar.add(  file   );
94        mBar.add(  edit   );
95        mBar.add(  color  );
96
97            //  Create and add menu items to menus
98
99            //  "File" menu
100       MenuItem clear  = new MenuItem( "Clear", new MenuShortcut(KeyEvent.VK_C
101       clear.addActionListener(f);
102       clear.setActionCommand( "clear" );
103       file.add(clear);
104
105       MenuItem quit  = new MenuItem( "Quit", new MenuShortcut(KeyEvent.VK_Q)
106       quit.addActionListener(f);
107       quit.setActionCommand( "quit" );
108       file.add(quit);
109
110           //  "Edit" menu
111       /*
112       MenuItem undo = new MenuItem( "Undo", new MenuShortcut(KeyEvent.VK_Z)
113       undo.addActionListener(f);
114       undo.setActionCommand( "undo" );
115       edit.add(undo);
116
117       ....
118
119       MenuItem paste  = new MenuItem( "Paste", new MenuShortcut(KeyEvent.VK_
120       paste.addActionListener(f);
121       paste.setActionCommand( "paste" );
122       edit.add(paste);
123       */
124
125       createMenuItem(  edit,   f, "Undo",   "undo",   KeyEvent.VK_Z );
126       createMenuItem(  edit,   f, "Cut",    "cut",    KeyEvent.VK_X );
127       createMenuItem(  edit,   f, "Copy",   "copy",   KeyEvent.VK_C );
128       createMenuItem(  edit,   f, "Paste", "paste", KeyEvent.VK_V );
129
130           //  "Color" menu
131       createMenuItem(  color, f, "Red",    "red",    KeyEvent.VK_R );
```

```
132            createMenuItem( color, f, "Green", "green", KeyEvent.VK_G );
133            createMenuItem( color, f, "Blue",  "blue",  KeyEvent.VK_B );
134        }
135
136        /** A utility function for creating and adding menu items to a menu.
137         */
138        protected static void createMenuItem( Menu m, ActionListener listener,
139            String label, String cmdString, int shortcut )
140        {
141            MenuItem mi = new MenuItem( label, new MenuShortcut(shortcut) );
142            mi.addActionListener( listener );
143            mi.setActionCommand( cmdString );
144            m.add( mi );
145        }
146 }
```

24 Event Handling for Macintosh and Windows Applications

There are many similarities in how events are processed in both Macintosh and Windows programs when compared to the original Java event handling model.

24.1 Example: Macintosh

```
1 /*   HelloMac.c
2  *
3  *   A simple "Hello" program with event handling.
4  *   Needless to say, the event handling is NOT complete.
5  *
6  *   Bruce M. Bolden
7  *   March 3, 1998
8  * -------------------------------------------------------------*/
9
10 #include <Types.h>
11 #include <Memory.h>
12 #include <Quickdraw.h>
13 #include <Fonts.h>
14 #include <Events.h>
15 #include <Menus.h>
16 #include <Windows.h>
17 #include <TextEdit.h>
18 #include <Dialogs.h>
19 #include <OSUtils.h>
20 #include <ToolUtils.h>
21
22      //   Constants
23 #define kAppleMenuID   100
24
25 #define kFileMenuID    101
26 #define kQuitMenuID      1
27
28 #define kEditMenuID    102
29
30      //   Globals
31 Boolean   gDone = false;        //   true when time to quit
32 Rect      gWindRect;            //   main window Rect
33
```

```
34 MenuHandle   gMenuDesk ;       //  menu handles
35 MenuHandle   gMenuFile ;
36 MenuHandle   gMenuEdit ;
37
38     //  Prototypes
39 void  Initialize ();
40 void  CreateMainWindow ();
41 void  SayHello ();
42
43 void  EventLoop ();
44
45 void  DoMouseDown( EventRecord *theEvent );
46 void  DoMenu( long menuCode );
47 void  DoAppleMenu( short itemID );
48 void  InitAppMenus ();
49
50
51
52 int  main ()
53 {
54     Initialize ();
55     InitAppMenus ();
56     CreateMainWindow ();
57
58     TextSize( kFontSize );
59
60     while (!gDone)            //  main event loop
61     {
62         EventLoop ();
63     }
64
65     return 0;
66 }
67
68
69 /*  Initialize ()
70  *
71  *  Initialize all the usual managers
72  */
73
74 void  Initialize ()
75 {
76     InitGraf(&qd.thePort);
77     InitFonts ();
78     InitWindows ();
```

```
 79     InitMenus();
 80     TEInit();
 81     InitDialogs(nil);
 82     InitCursor();
 83
 84     // To make the Random sequences truly random, we need to make the seed st
 85     // at a different number. An easy way to do this is to put the current t
 86     // and date into the seed. Only needed once, here in the initialization
 87
 88     GetDateTime( (unsigned long*)&qd.randSeed);
 89 }
 90
 91
 92 /*  CreateMainWindow
 93  *
 94  *  Make a new window for drawing in, and it must be a color window.
 95  *  The window is full screen size, made smaller to make it more visible.
 96  */
 97
 98 void CreateMainWindow()
 99 {
100     WindowPtr   mainWinPtr;
101
102     gWindRect = qd.screenBits.bounds;
103     InsetRect(&gWindRect, 50, 50);
104     mainWinPtr = NewCWindow(nil, &gWindRect, "\pHello", true, documentProc,
105                             (WindowPtr)-1, false, 0);
106
107     SetPort(mainWinPtr);    // set window to current graf port
108 }
109
110
111 /*  SayHello
112  *
113  *  Put "Hello!" in the window at some random location.
114  */
115
116 void SayHello()
117 {
118     int   x, y;
119     RGBColor  textColor;
120
121         // Make a random color for the message
122
123     textColor.red    = Random();
```

```
124      textColor.green  = Random();
125      textColor.blue   = Random();
126
127          //  Set this color as the new color to use in drawing
128
129      RGBForeColor (&textColor);
130
131          //  Generate a random location
132
133      x = Random();
134      y = Random();
135      x = ((x+32767) * gWindRect.bottom)/65536;
136      y = ((y+32767) * gWindRect.right)/65536;
137
138      MoveTo( x, y );
139      DrawString("\pHello!");
140 }
141
142
143 /*   EventLoop
144  *
145  *   Event processing
146  */
147
148 #define MIN_SLEEP            0L      /* value used by WaitNextEvent */
149 #define NIL_MOUSE_REGION     0L
150
151 void EventLoop()
152 {
153      char      c ;
154      EventRecord theEvent ;
155
156      WaitNextEvent( everyEvent, &theEvent, MIN_SLEEP, NIL_MOUSE_REGION );
157
158      switch( theEvent.what ) {
159      case keyDown:
160          if( theEvent.modifiers & cmdKey ) {
161              c = theEvent.message & charCodeMask ;
162              DoMenu( MenuKey(c) ) ;
163          }
164          break ;
165
166      case mouseDown:
167          DoMouseDown( &theEvent ) ;
168          break ;
```

```
169
170     case updateEvt:
171         //DoUpdate( theEvent.message ) ;
172         break ;
173
174     default:
175         SayHello();
176         break ;
177     }
178 }
179
180
181 /*  DoMouseDown
182  *
183  *  Handle the mouseDown events
184  */
185 void DoMouseDown( EventRecord *theEvent )
186 {
187     WindowRef    whichWindow;
188     short        thePart;
189     long         menuChoice;
190
191
192     thePart = FindWindow( theEvent->where, &whichWindow );
193
194     switch( thePart )
195     {
196     case inMenuBar:
197          menuChoice = MenuSelect( theEvent->where );
198          DoMenu( menuChoice );
199          break;
200
201     case inSysWindow:
202          SystemClick( theEvent, whichWindow );
203          break;
204
205     case inContent:
206          if( whichWindow != FrontWindow() ) {
207               SelectWindow(whichWindow);
208               /*DoEvent(event);*/ /* use this line for "do first click" */
209          }
210          else {
211               //(void)DoContent(whichWindow, theEvent);
212          }
213          break;
```

```
214
215     case inDrag:
216         if( whichWindow != FrontWindow() )
217             SelectWindow(whichWindow);
218         //else if( IsAppWindow(whichWindow) )
219         //  DragWindow( whichWindow, theEvent->where, &gDragRect );
220         break;
221
222     case inGrow:
223         if( whichWindow != FrontWindow() )
224             SelectWindow(whichWindow);
225         //else if( IsAppWindow(whichWindow) )
226         //  (void)DoGrow( whichWindow, theEvent );
227         break ;
228
229     case inGoAway:
230         if( TrackGoAway(whichWindow, theEvent->where) )
231         {
232             //if( DoCloseWindow( whichWindow ) )
233             //  CloseWindow( whichWindow );
234         }
235         break;
236
237     case inZoomIn:
238     case inZoomOut:
239         if( whichWindow != FrontWindow() )
240         {
241             SelectWindow(whichWindow);
242         }
243         //else
244         //  (void)DoZoom( whichWindow, thePart );
245         break;
246     }
247 }
248
249
250 /*  DoMenu
251  *
252  *  Handle menu selection
253  */
254
255 void DoMenu( long menuCode )
256 {
257     const short menuID = HiWord( menuCode );
258     const short itemID = LoWord( menuCode );
```

```
259
260     switch ( menuID )
261     {
262     case kAppleMenuID :
263         DoAppleMenu ( itemID ) ;
264         break ;
265
266     case kFileMenuID :
267         switch ( itemID )
268         {
269         case kQuitMenuID :
270             gDone = true ;
271             break ;
272         }
273         break ;
274
275     default :
276         break ;
277     }
278
279     HiliteMenu ( 0 ) ;
280 }
281
282
283 /*  DoAppleMenu
284  *
285  *  Handle Apple menu selection
286  */
287
288 void DoAppleMenu ( short itemID )
289 {
290     WindowPtr   currentPort ;
291     Str255  daName ;
292
293     GetMenuItemText ( gMenuDesk , itemID , daName ) ;
294     GetPort ( &currentPort ) ;
295     OpenDeskAcc ( daName ) ;
296     SetPort ( currentPort ) ;
297 }
298
299
300 /*  InitAppMenus
301  *
302  *  Initialize Application menu---normally done using resources !
303  */
```

```
304
305 void InitAppMenus()
306 {
307     gMenuDesk = NewMenu( kAppleMenuID, "\p\024" ) ;
308     AppendResMenu( gMenuDesk, 'DRVR' ) ;
309     InsertMenu( gMenuDesk, 0 ) ;
310
311     gMenuFile = NewMenu( kFileMenuID, "\pFile" ) ;
312     AppendMenu( gMenuFile, "\pQuit/Q" ) ;
313     InsertMenu( gMenuFile, 0 ) ;
314
315     gMenuEdit = NewMenu( kEditMenuID, "\pEdit" ) ;
316     AppendMenu( gMenuEdit, "\p(Undo;(-;(Cut;(Copy;(Paste;(Clear" ) ;
317     InsertMenu( gMenuEdit, 0 ) ;
318
319     DrawMenuBar();
320 }
```

24.2 Example: Windows

```
1 /*------------------------------------------------------------
2    HELLOWIN.C -- Displays "Hello, Windows 95!" in client area
3                  (c) Charles Petzold, 1996
4   ------------------------------------------------------------*/
5
6 #include <windows.h>
7
8 LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
9
10 int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
11                     PSTR szCmdLine, int iCmdShow)
12 {
13      static char szAppName[] = "HelloWin" ;
14      HWND        hwnd ;
15      MSG         msg ;
16      WNDCLASSEX  wndclass ;
17
18      wndclass.cbSize        = sizeof (wndclass) ;
19      wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
20      wndclass.lpfnWndProc   = WndProc ;
21      wndclass.cbClsExtra    = 0 ;
22      wndclass.cbWndExtra    = 0 ;
23      wndclass.hInstance     = hInstance ;
24      wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
25      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
26      wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
27      wndclass.lpszMenuName  = NULL ;
28      wndclass.lpszClassName = szAppName ;
29      wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
30
31      RegisterClassEx (&wndclass) ;
32
33      hwnd = CreateWindow (szAppName,         // window class name
34                     "The Hello Program",     // window caption
35                     WS_OVERLAPPEDWINDOW,     // window style
36                     CW_USEDEFAULT,           // initial x position
37                     CW_USEDEFAULT,           // initial y position
38                     CW_USEDEFAULT,           // initial x size
39                     CW_USEDEFAULT,           // initial y size
40                     NULL,                    // parent window handle
41                     NULL,                    // window menu handle
42                     hInstance,               // program instance handle
43                     NULL) ;                  // creation parameters
```

```
44
45       ShowWindow (hwnd, iCmdShow) ;
46       UpdateWindow (hwnd) ;
47
48       while (GetMessage (&msg, NULL, 0, 0))
49       {
50            TranslateMessage (&msg) ;
51            DispatchMessage (&msg) ;
52       }
53
54       return msg.wParam ;
55 }
56
57
58 LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
59                           WPARAM wParam, LPARAM lParam)
60 {
61       HDC         hdc ;
62       PAINTSTRUCT ps ;
63       RECT        rect ;
64
65       switch (iMsg)
66       {
67       case WM_CREATE :
68            PlaySound ("hellowin.wav", NULL, SND_FILENAME | SND_ASYNC) ;
69            return 0 ;
70
71       case WM_PAINT :
72            hdc = BeginPaint (hwnd, &ps) ;
73
74            GetClientRect (hwnd, &rect) ;
75
76            DrawText (hdc, "Hello, Windows 95!", -1, &rect,
77                           DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
78
79            EndPaint (hwnd, &ps) ;
80            return 0 ;
81
82       case WM_DESTROY :
83            PostQuitMessage (0) ;
84            return 0 ;
85       }
86
87       return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
88 }
```

25 Input and Output in Java

Input/output (I/O) techniques are generally not very exciting. However, without the ability to read and write data, applications (and some applets) would be very limited.

25.1 Input and Output Streams

In Java, an object from which bytes can be read is called an *input stream*. An object to which bytes can be written is called an *output stream*. These are implemented in the abstract classes `InputStream` and `OutputStream`.

Recall that the point of an *abstract* class is to provide a mechanism for factoring out the common behavior of classes to a higher level.

25.2 Input and Output Stream Processing

Since byte-oriented streams are inconvenient for processing information in Unicode (recall that Unicode uses two bytes per character), there is a separate hierarchy of classes for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. These classes have read and write operations that are based on 2-byte Unicode characters rather than on single-byte characters.

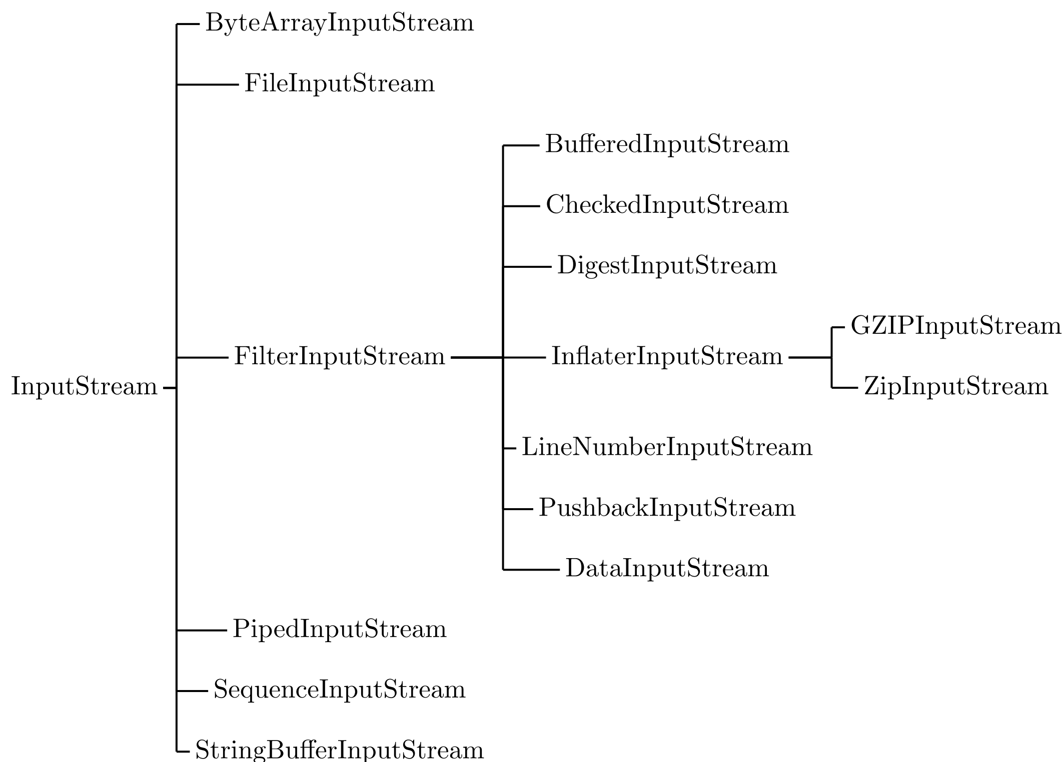
25.3 Input and Output Stream Hierarchies

Java has a *zoo* of 58 different stream types. The library designers claim that there is a good reason to give users a wide choice of stream types: it is supposed to reduce programming errors.

For example, in C, some people think it is a common mistake to send output to a file that was open only for reading. (Well, it is not that common, actually.) If you do this, the output is ignored at run time. In Java and C++, the compiler catches this kind of mistake because an `InputStream` (Java) or `istream` (C++) has no methods for output.[8]

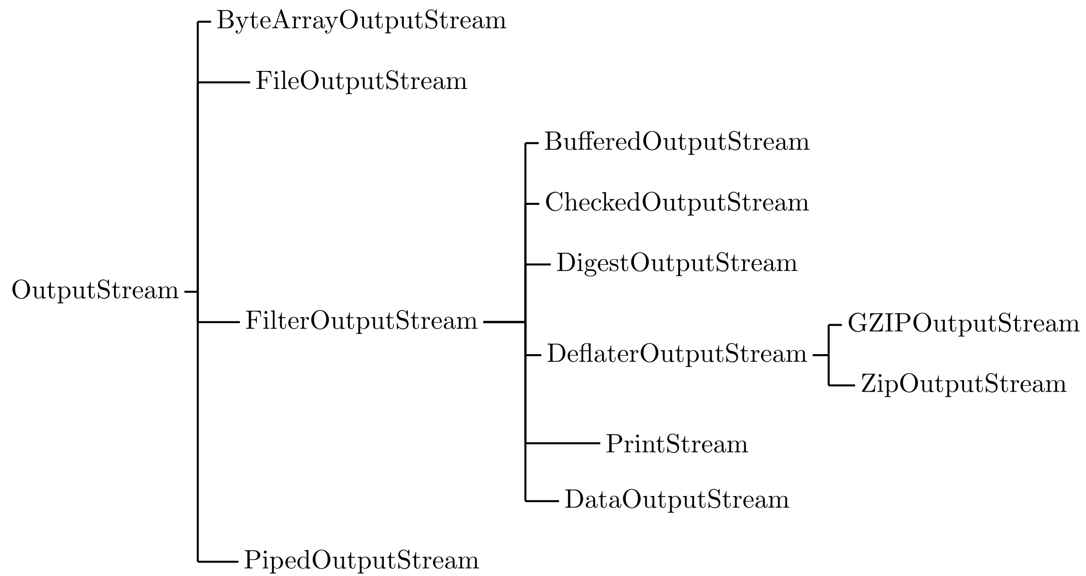
The following sections illustrate the stream type zoo. Note that there is some overlap between the Input and Output streams.

25.3.1 Input Stream Hierarchy

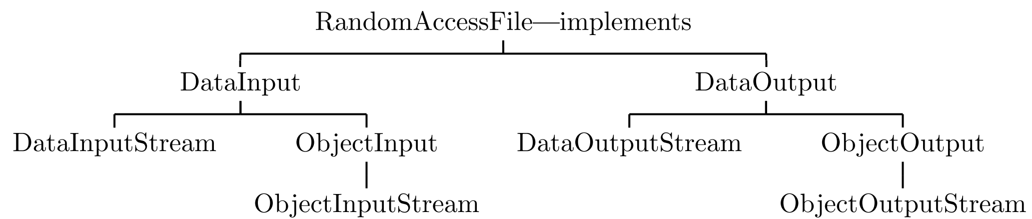


[8]Core Java, Volume II, Horstmann and Cornell, 1998

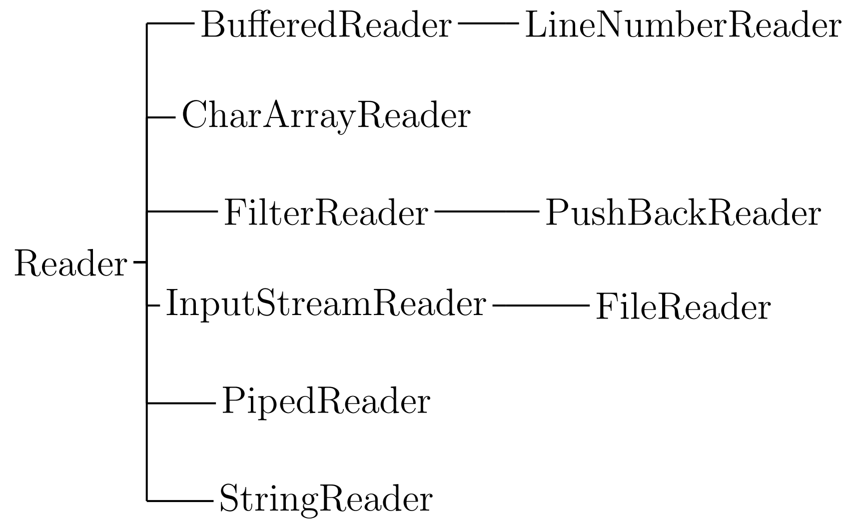
25.3.2 Output Stream Hierarchy



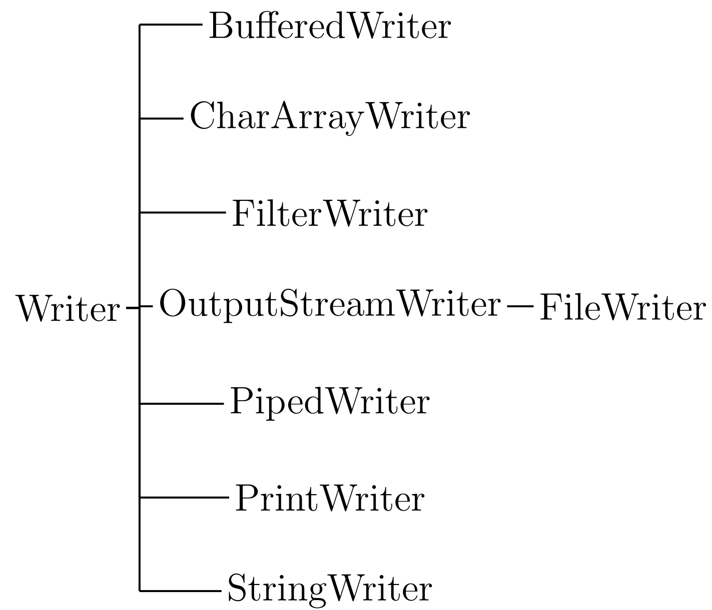
25.3.3 Connected I/O Stream Hierarchy



25.3.4 Reader Hierarchy



25.3.5 Writer Hierarchy



25.4 Reading and Writing Bytes

The `InputStream` class has an abstract method:

```
public abstract int read()  throws  IOException
```

This method reads one byte and returns the byte read, or -1 if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the `FileInputStream` class, this method reads one byte from a file. The `InputStream` class also has non-abstract methods to read an array of bytes or to skip a number of bytes. These methods call the abstract `read` method, so that subclasses only need to override one method.

Similarly, the `OutputStream` class defines the abstract method

```
public abstract void write(int b)  throws  IOException
```

which writes one byte to the output file.

Both the `read` and `write` methods can *block* a thread until the byte is actually read or written. This means if the byte cannot immediately be read from or written to (usually a busy network connection), Java suspends the thread containing this call. This gives other threads a chance to do useful work while the method is waiting for the stream to again become available.

25.4.1 Close Stream Objects

When finished reading or writing to a stream, close it (using `close`, because streams use operating system resources that are in limited supply. Closing an output stream also *flushes* the buffer used for the output stream: any characters that were

placed in a buffer so that they could be delivered as a larger packet are sent off. If you do not close a file, the last packet of bytes may never be delivered. Output can also be flushed manually using the `flush` method.

25.5 More Stream Classes

Java provides many stream classes derived from the basic I/O stream classes: `InputStream` and `OutputStream`. These classes let you work with data in the forms that you usually use rather than at the low, byte level. This is described in the next section.

25.6 Stream Filters

`FileInputStream` and `FileOutputStream` give you input and output streams attached to a disk file. You give the name or full pathname of the file in the constructor. For example,

```
FileInputStream fIn = new FileInputStream( "grades.dat" );
```

looks in the current directory for a file named `grades.dat`. You can also use a `File` object:

```
File f = new File( "grades.dat" );
FileInputStream fIn = new FileInputStream( f );
```

Just as the `FileInputStream` class has no methods to read numeric types, the `DataInputStream` class has no methods to get data from a file.

Java uses a clever mechanisim to separate two kinds of responsibilities. Some streams (such as `FileInputStream` and the input stream returned by the `openStream` method of the URL class) can retrieve bytes from files and other *exotic* locations.

Other streams (such as `DataInputStream` and the `PrintWriter` can assemble bytes into more useful data types. The Java programmer has to combine the two into what are often called *filtered streams* by feeding an existing stream to the constructor of another stream.

25.6.1 Stream Filter Example

For example, to be able to read numbers from a file, first create a `FileInputStream` and then pass it to the constructor of a `DataInputStream`.

```
FileInputStream fIn = new FileInputStream( "grades.dat" );
DataInputStream dIn = new DataInputStream( fIn );
double  r = dIn.readDouble();
```

The data input stream does not correspond to a new disk file. It accesses the data from the file attached to the file input stream, but it has a more capable interface.

25.6.2 Buffered Stream Filters

If you look back to the figures that illustrate the `Input` and `Output` stream hierarchy, specifically `FilterInputStream` and `FilterOutputStream`. You can combine their child classes into a new filtered stream to construct the streams you want. For example, by default, streams are not buffered. That is, every call to read contacts the operating system to ask it for another byte. If you want buffering *and* data input, you need to use the following (somewhat monstrous) sequence of constructors:

```
DataInputStream dIn = new DataInputStream(
    new BufferedInputStream(
```

```
            new FileInputStream( "grades.dat" ) ) );
```

Notice that `DataInputStream` is *last* in the chain of constructors because we want to use the `DataInputStream` methods, and we want *them* to use the buffered read method.

25.6.3 Example: Reading and Writing Bytes

```
//  byteRW.java

/**  Copy using FileInput/Output stream classes.
 */

public void FileStreamReadWrite()
{
    String inFileName = "test.in";
    String outFileName = "test2.out";

    FileInputStream fIS;
    FileOutputStream fOS;

    try {
       fIS = new FileInputStream(  inFileName  );
       fOS = new FileOutputStream( outFileName );

       int i = 0;

       while( i != -1 )
       {
          i = fIS.read();
         if( i != -1 )          //  remove this---see what happens
              fOS.write( i );
       }

       fIS.close();
       fOS.close();
    }
    catch (java.io.IOException ioe) {
        System.out.println( "IO error: " + ioe );
    }
}
```

25.7 Reading and Writing Text

25.7.1 Writing Text

For text output, you want to use a `PrintWriter`. A print writer can print strings and numbers in a text format. Just as a `DataOutputStream` has useful output methods, but no destination, a `PrintWriter` must be combined with a destination writer.

```
PrintWriter out = new PrintWriter(
    new FileWriter( "grades.dat" ) );
```

A print writer can also be combined with a destination (output) stream:

```
PrintWriter out = new PrintWriter(
       new FileOutputStream( "grades.dat" ) ) );
```

The `PrintWriter( OutputStream )` constructor automatically adds an `OutputStreamWriter` to convert Unicode characters to bytes in the stream.

25.7.2 Writing To A Print Writer

To write to a print writer, you use the same `print` and `println` methods that you used with `System.out`. You can use these methods to print numbers (`int, short, long, double`), characters, boolean values, strings *and* objects.

For example, the following code fragment:

```
String name = "Stan Smith";
double grade = 95;
out.print( name );
```

```
out.print( ' ' );
out.println( grade );
```

writes the characters

```
Stan Smith 95
```

to the stream `out`. The characters are converted to bytes and are put in the file `grades.dat`.

25.7.3 Reading Text

As you have seen:

- To write data in binary format, you use a `DataOutputStream`.
- To write in text format, you use a `PrintWriter`.

Therefore, you might expect that there is an analog to the `DataInputStream` that lets you read data in text format. Unfortunately, Java does not provide such a class. (That's why Horstmann and Cornell wrote the `Console` class.) The only game in town for processing text input is the `BufferedReader` class—it has a method, `readLine`, that lets you read a line of text. You need to combine a buffered reader with an input source.

25.7.4 Reading With A `BufferedReader`

```
BufferedReader in = new BufferedReader(
       new FileReader( "grades.dat" ) );
```

The `readLine` method returns `null` when no input is available. A typical input loop looks like:

```
String s;
while( (s = in.readLine()) != null)
{
     do something with s
}
```

25.7.5 More Reading With A `BufferedReader`

The `FileReader` class already converts bytes to Unicode characters. For other input sources, you need to use the `InputStreamReader`—unlike the `PrintWriter` class, there is no automatic convenience method to bridge the gap between bytes and Unicode characters.

```
BufferedReader in2 = new BufferedReader(
       new InputStreamReader( System.in ) );
BufferedReader in3 = new BufferedReader(
       new InputStreamReader( url.openStream() ) );
```

To read numbers from text input, you need to read a string first and then convert it.

```
String s = in.readLine();
double x = new Double(s).doubleValue();
```

25.7.6 Example: Buffered Reading and Writing

```
//  bufferRW.java

/**  Copy using Buffered Reader/Writer class.
 */
public static void BufferedFileStreamReadPrint()
{
      String inFileName = "test.in";
      String outFileName = "test3.out";

      FileReader fIS;
      FileWriter fW;

      try {
          fIS = new FileReader( inFileName  );
          BufferedReader inStream = new BufferedReader( fIS  );
          fW = new FileWriter(outFileName);
          PrintWriter outStream = new PrintWriter(
                  new BufferedWriter(fW));

          String line;

          while( (line = inStream.readLine()) != null)
          {
              outStream.println( line  );
          }

          inStream.close();
          outStream.close();
     }
     catch (java.io.IOException ioe) {
          System.out.println( "IO error: " + ioe  );
     }

}
```

25.8 References

Chapter 12, **Core Java 1.2**, Horstmann & Cornell

Hughes, *Reading textual data: Fun with Streams*, JavaWorld, April 1999

`http://www.javaworld.com/javaworld/jw-04-1999/jw-04-step.html`

26 Windows

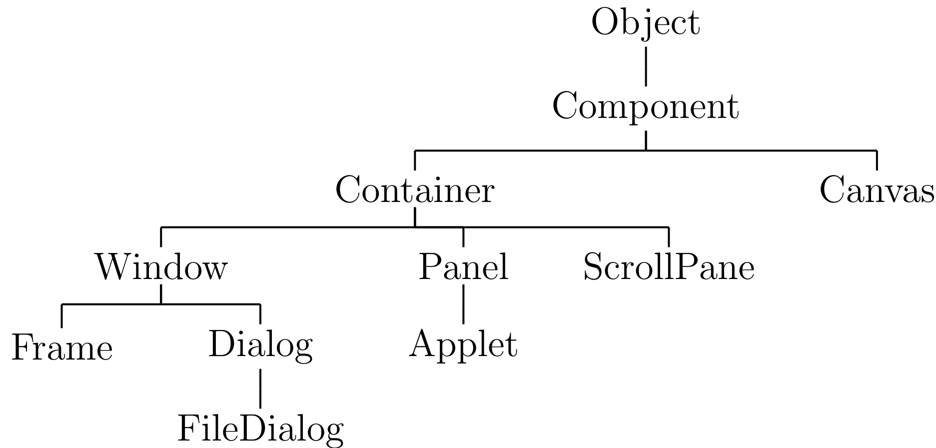
There are eight major types of windows in Java's AWT.

- `Frame`
- `Canvas`
- `Panel`
- `ScrollPane`
- `Applet`
- `Dialog`
- `FileDialog`
- `Window`

Some Windows are borderless (`Panel` and `ScrollPane`) and can only be placed in other windows. Other windows (e.g., `Frame` and `Dialog`) have borders and title bars and can be displayed anywhere on the screen. As we have seen, `Frame` is the base window for graphical applications. As we shall see, `Applet` is the starting point for Web-based applications (applets).

Most windows can contain other graphical components, but a few (`Canvas` `FileDialog`) cannot. Except for `ScrollPane`, all the windows that can contain other components have a `LayoutManager` that helps arrange the nested components. Except for `FileDialog`, all the windows receive mouse and keyboard events.

26.1 Component Class



A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

The Component class is the abstract superclass of the nonmenu-related AWT components. Class Component can also be extended directly to create a lightweight component. A lightweight component is a component that is not associated with a native opaque window (JDK 1.1^{+} feature).

26.1.1 Frame

A `Frame` is a top-level window with a title and a border. The default layout for a frame is `BorderLayout`. Frames are capable of generating the following types of window events: `WindowOpened`, `WindowClosing`, `WindowClosed`, `WindowIconified`, `WindowDeiconified`, `WindowActivated`, `WindowDeactivated`.

26.1.2 Canvas

A `Canvas` component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.

An application must subclass the `Canvas` class in order to get useful functionality such as creating a custom component. The `paint` method must be overridden in order to perform custom graphics on the canvas.

26.1.3 Panel

`Panel` is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.

The default layout manager for a panel is the `FlowLayout` layout manager.

26.1.4 Applet

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.

The `Applet` class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The `Applet` class provides a standard interface between applets and their environment.

26.1.5 Dialog

A class that produces a dialog - a window that takes input from the user. The default layout for a dialog is `BorderLayout`.

Dialogs are capable of generating the following window events: `WindowOpened`, `WindowClosing`, `WindowClosed`, `WindowActivated`, `WindowDeactivated`.

26.1.6 FileDialog

The `FileDialog` class displays a dialog window from which the user can select a file.

Since it is a modal dialog, when the application calls its `show` method to display the dialog, it blocks the rest of the application until the user has chosen a file.

How to tell a file about another drive.

26.1.7 ScrollPane

A container class which implements automatic horizontal and/or vertical scrolling for a single child component. The display policy for the scrollbars can be set to:

1. as needed: scrollbars created and shown only when needed by scrollpane
2. always: scrollbars created and always shown by the scrollpane
3. never: scrollbars never created or shown by the scrollpane

The state of the horizontal and vertical scrollbars is represented by two objects (one for each dimension) which implement the `Adjustable` interface. The API provides methods to access those objects such that the attributes on the `Adjustable` object (such as unitIncrement, value, etc.) can be manipulated.

Certain adjustable properties (`minimum`, `maximum`, `blockIncrement`, and `visibleAmount`) are set internally by the scrollpane in accordance with the geometry of the scrollpane and its child and these should not be set by programs using the scrollpane.

If the scrollbar display policy is defined as "never", then the scrollpane can still be programmatically scrolled using the `setScrollPosition()` method and the scrollpane will move and clip the child's contents appropriately. This policy is useful if the program needs to create and manage its own adjustable controls.

The placement of the scrollbars is controlled by platform-specific properties set by the user outside of the program.

The initial size of this container is set to 100x100, but can be reset using `setSize()`.

Insets are used to define any space used by scrollbars and any borders created by the scroll pane. `getInsets()` can be used to get the current value for the insets. If the value of `scrollbarsAlwaysVisible` is false, then the value of the insets will change dynamically depending on whether the scrollbars are currently visible or not.

26.2 Canvas

A `Canvas` is the simplest window type in Java. It cannot contain any GUI controls or nested windows. NOTE: It is not a standalone window—it must be placed in an existing window [9].

[9] Marty Hall, Core Web Programming, Prentice-Hall, 1998

Initial Output from CircleTest program

Output from CircleTest program after resizing window

```
1 /*  Circle.java
2  *
3  *  Defines a circular class object using a Canvas.
4  *  Variation of an example in:
5  *  Marty Hall, Core Web Programming, Prentice-Hall, 1998
6  *
7  *  Bruce M. Bolden
8  *  March 22, 1998
9  *  http://www.cs.uidaho.edu/~bruceb/
10 */
11
12 import java.awt.Canvas;
13 import java.awt.Color;
14 import java.awt.Graphics;
15
16 /**  Parent class for all circular objects.   */
17
18 final public class Circle extends Canvas
19 {
20     private static final int DEFAULT_RADIUS = 20;
21
22     /**  Constructs a circle with color, c, and radius, r.  */
23     public Circle( Color c, int r ) {
24         setForeground( c );
25         setSize( 2*r, 2*r );          //  resize()  in Java 1.02
26     }
27
28     /**  paint the circle */
29     public void paint( Graphics g )
30     {
31         g.fillOval( 0, 0, getSize().width, getSize().height );
32     }
33
34     /**  Set the center of the circle (account for canvas
35           dimentsions). */
36     public void setCenter( int x, int y )
37     {
38         int h = getSize().height;     //  size()  in Java 1.02
39         int w = getSize().width;
40         setLocation( x - w/2, y - w/2 ); // move() in Java 1.02
41     }
42 }
```

```
 1 /*   CircleTest.java
 2  *
 3  *   From Core Java I, Horstmann and Cornell
 4  *
 5  *   March 22, 1998
 6  */
 7
 8 import java.awt.*;
 9 import java.awt.event.*;
10 import corejava.*;
11
12 public class CircleTest
13     extends CloseableFrame
14 {
15     public static void main( String[] args )
16     {
17         CircleTest f = new CircleTest();
18         f.show();
19     }
20
21     public CircleTest()
22     {
23         setLayout( new FlowLayout() );
24
25             // Create circles and add them
26         add( new Circle( Color.red,   30 ) );
27         add( new Circle( Color.green, 40 ) );
28         add( new Circle( Color.blue,  50 ) );
29     }
30 }
```

In Java 1.02, windows and GUI controls are rectangular and opaque. This is illustrated in the following example.

Output from CircleTest2 program

```
1 /*   CircleTest2.java
2  *
3  *   Variation of an example in:
4  *   Marty Hall, Core Web Programming, Prentice-Hall, 1998
5  *
6  *   Bruce M. Bolden
7  *   March 22, 1998
8  */
9
10 import java.awt.*;
11 import java.awt.event.*;
12 import corejava.*;
13
14 public class CircleTest2
15     extends CloseableFrame
16 {
17     static final int CIRCLE_RADIUS = 30;
18
19     public static void main( String[] args )
20     {
21         CircleTest2 f = new CircleTest2();
22         f.show();
23     }
24
25     public CircleTest2()
26     {
27         setLayout( null );
28
29             // Create circles and add them
30         Circle rCircle = new Circle( Color.red,   CIRCLE_RADIUS );
31         add( rCircle );
32         rCircle.setCenter( CIRCLE_RADIUS, CIRCLE_RADIUS );
33
34         Circle gCircle = new Circle( Color.green, CIRCLE_RADIUS );
35         add( gCircle );
36         gCircle.setCenter( 2*CIRCLE_RADIUS, 2*CIRCLE_RADIUS );
37
38         Circle bCircle = new Circle( Color.blue,  CIRCLE_RADIUS );
39         add( bCircle );
40         bCircle.setCenter( 3*CIRCLE_RADIUS, 3*CIRCLE_RADIUS );
41
42             /*  reverse direction
43              *
44              *  Note that we must pass our object to our circle
45              *  "creation" method.
```

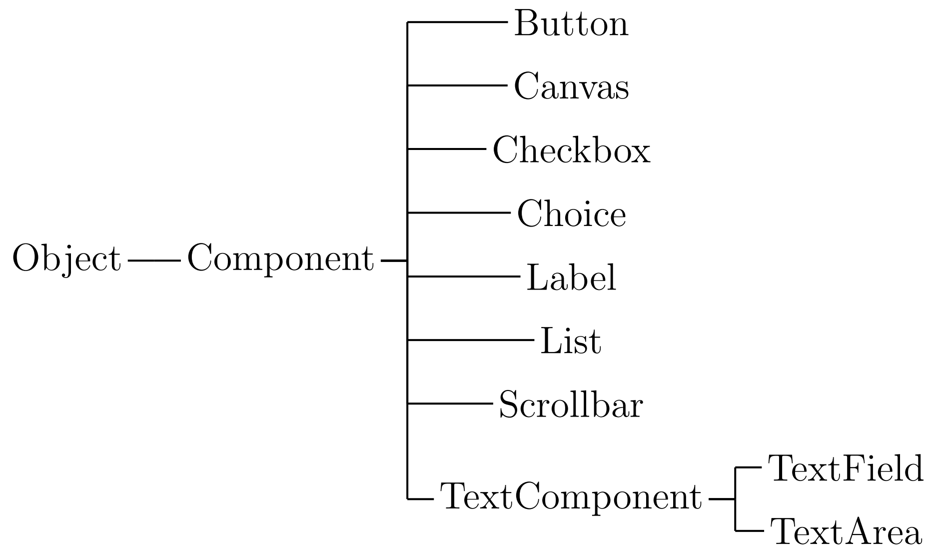
```
46              */
47
48          genCircle2( this );
49
50          //genCircle2a();
51      }
52
53      static void genCircle2( CircleTest2 cObj )
54      {
55          Circle bCircle2 = new Circle( Color.blue,  CIRCLE_RADIUS );
56          cObj.add( bCircle2 );
57          bCircle2.setCenter( 6*CIRCLE_RADIUS, 3*CIRCLE_RADIUS );
58
59          Circle gCircle2 = new Circle( Color.green, CIRCLE_RADIUS );
60          cObj.add( gCircle2 );
61          gCircle2.setCenter( 7*CIRCLE_RADIUS, 2*CIRCLE_RADIUS );
62
63          Circle rCircle2 = new Circle( Color.red,   CIRCLE_RADIUS );
64          cObj.add( rCircle2 );
65          rCircle2.setCenter( 8*CIRCLE_RADIUS, CIRCLE_RADIUS );
66      }
67
68      /*  Uncomment to see error message
69      static void genCircle2a( )
70      {
71          Circle bCircle2 = new Circle( Color.blue,  CIRCLE_RADIUS );
72
73          //  Error:
74          //  Can''t make static reference to method java.awt.Component
75          //    add(java.awt.Component) in class java.awt.Container.
76          //    CircleTest2.java line 70     add( bCircle3 );
77
78          add( bCircle2 );
79          bCircle2.setCenter( 5*CIRCLE_RADIUS, 4*CIRCLE_RADIUS );
80      }
81      */
82 }
```

27 Introduction to Components

The standard User Interface components that are not derived from `Containers` are described in this section.

27.1 Component Class Inheritance Hierarchy

The Component Class inheritance hierarchy is illustrated below.



27.2 Component Example

All Components

All Components

27.3 Component Example Source Code

```
1 // This example is from the book _Java in a Nutshell_ by David Flanagan.
2 // Written by David Flanagan. Copyright (c) 1996 O'Reilly & Associates.
3 // You may study, use, modify, and distribute this example for any purpose.
4 // This example is provided WITHOUT WARRANTY either expressed or implied.
5
6 import java.awt.*;
7
8 public class AllComponents extends Frame
9 {
10     MenuBar menubar;                    // the menubar
11     Menu file, help;                    // menu panes
12     Button okay, cancel;                // buttons
13     List list;                          // A list of choices
14     Choice choice;                      // A menu of choices
15     CheckboxGroup checkbox_group;       // A group of button choices
16     Checkbox[] checkboxes;              // the buttons to choose from
17     TextField textfield;                // One line of text input
18     TextArea  textarea;                 // A text window
19     ScrollableScribble scribble;        // An area to draw in.
20     FileDialog file_dialog;
21
22     Panel panel1, panel2 ;      // Sub-containers for all this stuff.
23     Panel buttonpanel;
24
25     // The layout manager for each of the containers.
26     GridBagLayout gridbag = new GridBagLayout();
27
28
29     public static void main(String[] args)
30     {
31         Frame f = new AllComponents("AWT Demo");
32         // We should call f.pack() here.  But its buggy.
33         f.setSize(450, 475);     //  was  resize()
34         f.show();
35     }
36
37     public AllComponents(String title)
38     {
39         super(title);
40
41         // Create the menubar.  Tell the frame about it.
42         menubar = new MenuBar();
43         this.setMenuBar(menubar);
```

```
44          // Create the file menu.  Add two items to it.  Add to menubar.
45          file = new Menu("File");
46          file.add(new MenuItem("Open"));
47          file.add(new MenuItem("Quit"));
48          menubar.add(file);
49          // Create Help menu; add an item; add to menubar
50          help = new Menu("Help");
51          help.add(new MenuItem("About"));
52          menubar.add(help);
53          // Display the help menu in a special reserved place.
54          menubar.setHelpMenu(help);
55
56          // Create pushbuttons
57          okay = new Button("Okay");
58          cancel = new Button("Cancel");
59
60          // Create a menu of choices
61          choice = new Choice();
62          choice.addItem("red");
63          choice.addItem("green");
64          choice.addItem("blue");
65
66          // Create checkboxes, and group them.
67          checkbox_group = new CheckboxGroup();
68          checkboxes = new Checkbox[3];
69          checkboxes[0] = new Checkbox("vanilla", checkbox_group, false);
70          checkboxes[1] = new Checkbox("chocolate", checkbox_group, true);
71          checkboxes[2] = new Checkbox("strawberry", checkbox_group, false);
72
73          // Create a list of choices.
74          list = new List(4, true);
75          list.addItem("Java");
76          list.addItem("C++");
77          list.addItem("Lisp");
78          list.addItem("Modula-3");
79          list.addItem("Object Pascal");
80          list.addItem("Simula");
81          list.addItem("Smalltalk");
82
83          // Create a one-line text field, and multi-line text area.
84          textfield = new TextField(15);
85          textarea = new TextArea(6, 40);
86          textarea.setEditable(false);
87
88          // Create a scrolling canvas to scribble in.
```

```
89          scribble = new ScrollableScribble();
90
91          // Create a file selection dialog box
92          file_dialog = new FileDialog(this, "Open␣File", FileDialog.LOAD);
93
94          // Create a Panel to contain all the components along the
95          // left hand side of the window. Use a GridBagLayout for it.
96          panel1 = new Panel();
97          panel1.setLayout(gridbag);
98
99          // Use several versions of the constrain() convenience method
100         // to add components to the panel and to specify their
101         // GridBagConstraints values.
102         constrain(panel1, new Label("Name:"), 0, 0, 1, 1);
103         constrain(panel1, textfield, 0, 1, 1, 1);
104         constrain(panel1, new Label("Favorite␣color:"), 0, 2, 1, 1,
105               10, 0, 0, 0);
106         constrain(panel1, choice, 0, 3, 1, 1);
107         constrain(panel1, new Label("Favorite␣flavor:"), 0, 4, 1, 1,
108               10, 0, 0, 0);
109         constrain(panel1, checkboxes[0], 0, 5, 1, 1);
110         constrain(panel1, checkboxes[1], 0, 6, 1, 1);
111         constrain(panel1, checkboxes[2], 0, 7, 1, 1);
112         constrain(panel1, new Label("Favorite␣languages:"), 0, 8, 1, 1,
113               10, 0, 0, 0);
114         constrain(panel1, list, 0, 9, 1, 3, GridBagConstraints.VERTICAL,
115               GridBagConstraints.NORTHWEST, 0.0, 1.0, 0, 0, 0, 0);
116
117         // Create a panel for the items along the right side.
118         // Use a GridBagLayout, and arrange items with constrain(), as above.
119         panel2 = new Panel();
120         panel2.setLayout(gridbag);
121
122         constrain(panel2, new Label("Messages"), 0, 0, 1, 1);
123         constrain(panel2, textarea, 0, 1, 1, 3, GridBagConstraints.HORIZONTAL
124               GridBagConstraints.NORTH, 1.0, 0.0, 0, 0, 0, 0);
125         constrain(panel2, new Label("Diagram"), 0, 4, 1, 1, 10, 0, 0, 0);
126         constrain(panel2, scribble, 0, 5, 1, 5, GridBagConstraints.BOTH,
127               GridBagConstraints.CENTER, 1.0, 1.0, 0, 0, 0, 0);
128
129         // Do the same for the buttons along the bottom.
130         buttonpanel = new Panel();
131         buttonpanel.setLayout(gridbag);
132         constrain(buttonpanel, okay, 0, 0, 1, 1, GridBagConstraints.NONE,
133               GridBagConstraints.CENTER, 0.3, 0.0, 0, 0, 0, 0);
```

```
134         constrain(buttonpanel, cancel, 1, 0, 1, 1, GridBagConstraints.NONE,
135                GridBagConstraints.CENTER, 0.3, 0.0, 0, 0, 0, 0);
136
137         // Finally, use a GridBagLayout to arrange the panels themselves
138         this.setLayout(gridbag);
139         // And add the panels to the toplevel window
140         constrain(this, panel1, 0, 0, 1, 1, GridBagConstraints.VERTICAL,
141                GridBagConstraints.NORTHWEST, 0.0, 1.0, 10, 10, 5, 5);
142         constrain(this, panel2, 1, 0, 1, 1, GridBagConstraints.BOTH,
143                GridBagConstraints.CENTER, 1.0, 1.0, 10, 10, 5, 10);
144         constrain(this, buttonpanel, 0, 1, 2, 1, GridBagConstraints.HORIZONTAL
145                GridBagConstraints.CENTER, 1.0, 0.0, 5, 0, 0, 0);
146     }
147
148     public void constrain(Container container, Component component,
149                   int grid_x, int grid_y, int grid_width, int grid_height,
150                   int fill, int anchor, double weight_x, double weight_y,
151                   int top, int left, int bottom, int right)
152     {
153         GridBagConstraints c = new GridBagConstraints();
154         c.gridx = grid_x; c.gridy = grid_y;
155         c.gridwidth = grid_width; c.gridheight = grid_height;
156         c.fill = fill; c.anchor = anchor;
157         c.weightx = weight_x; c.weighty = weight_y;
158         if (top+bottom+left+right > 0)
159             c.insets = new Insets(top, left, bottom, right);
160
161         ((GridBagLayout)container.getLayout()).setConstraints(component, c);
162         container.add(component);
163     }
164
165     public void constrain(Container container, Component component,
166                   int grid_x, int grid_y, int grid_width, int grid_height)
167     {
168         constrain(container, component, grid_x, grid_y,
169                grid_width, grid_height, GridBagConstraints.NONE,
170                GridBagConstraints.NORTHWEST, 0.0, 0.0, 0, 0, 0, 0);
171     }
172
173     public void constrain(Container container, Component component,
174                   int grid_x, int grid_y, int grid_width, int grid_height,
175                   int top, int left, int bottom, int right)
176     {
177         constrain(container, component, grid_x, grid_y,
178                grid_width, grid_height, GridBagConstraints.NONE,
```

```
179                    GridBagConstraints.NORTHWEST,
180                    0.0, 0.0, top, left, bottom, right);
181       }
182 }
```

28 Components

The standard User Interface components that are not derived from `Containers` are described in this section. We have discussed a Buttons and Canvases in detail previously.

28.1 Button

Buttons are useful for single actions. If the user clicks a button, they expect something to happen (e.g., dismissal of a dialog).

28.2 Canvas

Canvases allow drawing to be done without interfering with other components.

Recall that:

- the `Canvas` class must be subclassed in order to get useful functionality such as creating a custom component.
- the `paint` method must be overridden in order to perform custom graphics on the canvas.

28.3 Checkbox

Checkboxes allow a user to select zero or more options. Keeping track of the *states* associated with check boxes can be very tedious.

```
final int N_COLORS = 3;
Checkbox colorCheckbox[] = new Checkbox[N_COLORS];
colorCheckbox[0] = new Checkbox( "red"   );
colorCheckbox[1] = new Checkbox( "green" );
```

```
colorCheckbox[2] = new Checkbox( "blue"  );

        //  create event handler
ItemListener colorCheckBoxListener = new ItemListener()
{
    public void itemStateChanged( ItemEvent e )
    {
        String theColor =
            ((Checkbox)e.getItemSelectable()).getLabel() ;
        drawingArea.showColor( theColor );
        textArea.append( "colorChoice: " + theColor + "\n" );
    }
};

    //  add components to panel
panelCheck.add( circleInfoLabel );

for( int i = 0 ; i < N_COLORS ; i++ )
{
    colorCheckbox[i].addItemListener( colorCheckBoxListener );
    panelCheck.add( colorCheckbox[i] );
}
```

28.4 Checkbox Groups (radio buttons)

Checkbox groups allow a user to select a single option. Keeping track of the *states* associated with a checkbox group can be tedious.

28.5 Choice

Choices (drop-down lists) allow a user to select one specific option from a finite number of options. Efficient use of space. Usually, need a `Label` to help explain the choices.

```
Choice choice = new Choice();
choice.addItem("red");
choice.addItem("green");
choice.addItem("blue");
panelChoice.add( choice );

    //  create event handler
ItemListener colorChoiceListener = new ItemListener()
{
    public void itemStateChanged( ItemEvent e )
    {
        //drawingArea.showColor( (String)e.getItem() );
        drawingArea.setBackgroundColor( (String)e.getItem() );
        textArea.append( "colorChoice: " + e.getItem() + "\n" );
    }
};

choice.addItemListener( colorChoiceListener );
```

28.6 Label

A simple string (multiple lines) to help explain the meaning of a component to the user.

Creating a `Label`

```
Label circleInfoLabel = new Label( "Show:" );
```

28.7 List

A list displays multiple items in a scrollable box. Generally, more than one item is visible at a time, so Lists take up more screen space than Choices. Another difference is that multiple selections made be made from a list box.

28.8 TextField

A `TextField` can be used to retrieve/display information. Common uses are reading values or file names (URLs) for later use.

28.9 TextArea

`TextArea`s can be used to display a considerable amount of information. Common uses are displaying the contents of files or status information.

Creating a `TextArea`

```
TextArea textArea = new TextArea( 6, 35 );
```

28.10 Example

Initial Display

Display after changing settings

```
 1 /*   UIDemo.java
 2  *
 3  *   A program to illustrate some basic UI components (controls).
 4  *
 5  *   Bruce M. Bolden
 6  *   April 5, 1998
 7  *   http://www.cs.uidaho.edu/~bruceb/
 8  */
 9
10 import java.awt.*;
11 import java.awt.event.*;
12 import corejava.*;
13
14 public class UIDemo extends CloseableFrame
15     implements ActionListener
16 {
17     static DrawingCanvas drawingArea;
18     static TextArea textArea;            // A text window
19
20
21     public static void main(String[] args)
22     {
23         Frame f = new UIDemo("UI Demo");
24
25         f.setSize(350, 300);
26         f.show();
27     }
28
29     public void paint( Graphics g )
30     {
31         drawingArea.doUpdate();
32     }
33
34     public void actionPerformed( ActionEvent e )
35     {
36         /*
37               Dummy method
38         */
39
40         /*
41         if( e.getActionCommand().equals( "red" ) )
42             drawingArea.showColor( "red" );
43         else if( e.getActionCommand().equals( "green" ) )
44             drawingArea.showColor( "green" );
45         else if( e.getActionCommand().equals( "blue" ) )
```

```
46              drawingArea.showColor( "blue" );
47          */
48      }
49
50      public UIDemo(String title)
51      {
52          setLayout( new BorderLayout() );
53
54          // Control/Display regions
55          Panel  panelCheck  = new Panel();
56          Panel  panelChoice = new Panel();
57          Panel  panelDraw   = new Panel();
58
59              // use grid layout to place things correctly
60          panelChoice.setLayout( new GridLayout(8,1) );
61
62          // Drawing/Information display
63          drawingArea = new DrawingCanvas();
64          textArea    = new TextArea( 6, 35 );
65          panelDraw.add( drawingArea );
66          panelDraw.add( textArea );
67
68          // Color check boxes (display circles)
69          Label circleInfoLabel = new Label( "Show:" );
70
71          final int N_COLORS = 3;
72          Checkbox colorCheckbox[] = new Checkbox[N_COLORS];
73          colorCheckbox[0] = new Checkbox( "red"   );
74          colorCheckbox[1] = new Checkbox( "green" );
75          colorCheckbox[2] = new Checkbox( "blue"  );
76
77              // create event handler
78          ItemListener colorCheckBoxListener = new ItemListener()
79          {
80              public void itemStateChanged( ItemEvent e )
81              {
82                  String theColor =
83                      ((Checkbox)e.getItemSelectable()).getLabel() ;
84                      // set color and force redraw
85                  drawingArea.showColor( theColor );
86                  drawingArea.doUpdate();
87                  textArea.append( "colorChoice: " + theColor + "\n" );
88              }
89          };
90
```

```
 91             // add components and event handler to panel
 92         panelCheck.add( circleInfoLabel );
 93 
 94         for ( int i = 0 ; i < N_COLORS ; i++ )
 95         {
 96             colorCheckbox[i].addItemListener( colorCheckBoxListener );
 97             panelCheck.add( colorCheckbox[i] );
 98         }
 99 
100         // Create a menu for background choices
101         Label bgInfoLabel = new Label( "Background:" );
102         Choice choice = new Choice();
103         choice.addItem( "white" );
104         choice.addItem( "red"   );
105         choice.addItem( "green" );
106         choice.addItem( "blue"  );
107         panelChoice.add( bgInfoLabel );
108         panelChoice.add( choice );
109 
110             // create event handler
111         ItemListener colorChoiceListener = new ItemListener()
112         {
113             public void itemStateChanged( ItemEvent e )
114             {
115                     // set color and force redraw
116                 //drawingArea.showColor( (String)e.getItem() );
117                 drawingArea.setBackgroundColor( (String)e.getItem() );
118                 drawingArea.doUpdate();
119                 textArea.append( "Background␣color:␣" + e.getItem() + "\n" );
120             }
121         };
122 
123         choice.addItemListener( colorChoiceListener );
124 
125         // add panels to the frame
126         add( panelCheck,  "South"  );
127         add( panelChoice, "East"   );
128         add( panelDraw,   "Center" );
129     }
130 }
131 
132 
133 class DrawingCanvas extends Canvas
134 {
135     final int dcWidth  = 250;
```

```
136     final int dcHeight = 120;
137
138     boolean  showRed   = false;
139     boolean  showGreen = false;
140     boolean  showBlue  = false;
141
142     Color  bgColor = Color.white;     // background color
143
144     DrawingCanvas()
145     {
146         setSize( dcWidth, dcHeight );
147     }
148
149     /** force update by calling repaint(). */
150     public void doUpdate()
151     {
152         repaint();
153     }
154
155     /** display selected circles    */
156     public void paint( Graphics g )
157     {
158         setBackground( bgColor );
159         if( showRed )
160         {
161             g.setColor( Color.red );
162             g.fillOval( 25, 20, 50, 50 );
163         }
164
165         if( showGreen )
166         {
167             g.setColor( Color.green );
168             g.fillOval( 100, 20, 50, 50 );
169         }
170
171         if( showBlue )
172         {
173             g.setColor( Color.blue );
174             g.fillOval( 175, 20, 50, 50 );
175         }
176     }
177
178     /** set circles to be displayed   */
179     public void showColor( String s )
180     {
```

```
181            if( s.equals( "red" ) )
182                showRed = showRed ? false : true;
183            else if( s.equals( "green" ) )
184                showGreen = showGreen ? false : true;
185            else if( s.equals( "blue" ) )
186                showBlue = showBlue ? false : true;
187        }
188
189        /** set canvas background */
190        public void setBackgroundColor( String s )
191        {
192            if( s.equals( "red" ) )
193                bgColor = Color.red;
194            else if( s.equals( "green" ) )
195                bgColor = Color.green;
196            else if( s.equals( "blue" ) )
197                bgColor = Color.blue;
198            else
199                bgColor = Color.white;
200
201        }
202 }
```

29 Layout Managers

Layout managers available in Java 1.1x:

- Flow Layout
- Border Layout
- Card Layout
- Grid Layout
- Grid Bag Layout
- Custom Layout Managers

29.1 Flow Layout

This is the simplest layout manager. Flow layouts are the default layout for panels. The flow layout manager adds the components horizontally until there is no more room and then starts a new line of components.

By default, components are centered in a container. To use another alignment style, specify `LEFT` or `RIGHT` in the constructor of the `FlowLayout` object.

Nested Buttons

Nested Buttons (Orange Selected)

```
 1 /*   NestedButtonTest.java
 2  *
 3  *   Illustrates the use of background coloring and nested
 4  *   components.
 5  *
 6  *   Derived from Example 6-1 in
 7  *   "Java Examples in a Nutshell", David Flanagan,
 8  *   O'Reilly & Associates, 1997
 9  *
10  *   Frame -- panel1 -- button1
11  *     |          |
12  *     |          |-- panel2 -- button2
13  *     |          |
14  *     |          |-- panel3 -- button3
15  *     |          |
16  *     |          |-- panel4 -- button4
17  *     |                  |
18  *     |--  button6       |--  button5
19  *
20  *   March 29, 1998
21  */
22
23 import java.awt.*;
24 import java.awt.event.*;
25 //import corejava.*;
26
27 public class NestedButtonTest
28     extends CloseableFrame
29         implements ActionListener
30 {
31     public static void main( String[] args )
```

```
32      {
33          NestedButtonTest f = new NestedButtonTest();
34          f.show();
35      }
36
37      public NestedButtonTest()
38      {
39          setLayout( new FlowLayout() );
40          this.setBackground( Color.white );
41          this.setFont( new Font( "Dialog", Font.BOLD, 24 ) );
42
43              // Create buttons and register them
44          Button redButton     = new Button( "Red"     );
45          Button orangeButton  = new Button( "Orange"  );
46          Button yellowButton  = new Button( "Yellow"  );
47          Button greenButton   = new Button( "Green"   );
48          Button blueButton    = new Button( "Blue"    );
49          Button magentaButton = new Button( "Magenta" );
50
51          Panel p1 = new Panel();
52          p1.add( redButton );
53          p1.setBackground( Color.red );
54          this.add( p1 );
55          redButton.addActionListener( this );
56
57          Panel p2 = new Panel();
58          p2.add( orangeButton );
59          p2.setBackground( Color.orange );
60          p1.add( p2 );
61          orangeButton.addActionListener( this );
62
63          Panel p3 = new Panel();
64          p3.add( yellowButton );
65          p3.setBackground( Color.yellow );
66          p2.add( p3 );
67          yellowButton.addActionListener( this );
68
69          Panel p4 = new Panel();
70          p4.add( greenButton );
71          p4.setBackground( Color.green );
72          p1.add( p4 );
73          greenButton.addActionListener( this );
74
75          Panel p5 = new Panel();
76          p5.add( blueButton );
```

```
77          p5.setBackground( Color.blue );
78          p1.add( p5 );
79          blueButton.addActionListener( this );
80
81          Panel p6 = new Panel();
82          p6.add( magentaButton );
83          p6.setBackground( Color.magenta );
84          this.add( p6 );
85          magentaButton.addActionListener( this );
86      }
87
88      public void actionPerformed( ActionEvent e )
89      {
90          Color c = Color.black;
91
92          String arg = e.getActionCommand();
93              // Set color based upon button pressed
94          if(      arg.equals("Red"     ) ) c = Color.red;
95          else if( arg.equals("Orange"  ) ) c = Color.orange;
96          else if( arg.equals("Yellow"  ) ) c = Color.yellow;
97          else if( arg.equals("Green"   ) ) c = Color.green;
98          else if( arg.equals("Blue"    ) ) c = Color.blue;
99          else if( arg.equals("Magenta" ) ) c = Color.magenta;
100
101         setBackground( c );
102         repaint();
103     }
104 }
```

29.2 Border Layout

The border layout is also fairly simple. Border layouts are the default for frames and other windows. The border layout divides the area to be laid out into five areas, called `North`, `South`, `East`, `West`, and `Center`, as illustrated below.

Border Layout Configuration

The borders are laid out first, then the remaining space is allocated to the center. When the container is resized, the thickness of the borders are unchanged, but the center area changes its size.

Components are added by specifying a string (case sensitive) that describes the region in which the object should be placed. Not all positions must be specified.

Border Layout Configuration

Border Layout Configuration—After Resizing

```
1 /*   Border.java
2  */
3
4 import java.awt.*;
5 import corejava.*;
6
7 public class Border extends CloseableFrame
8 {
9     final static int FRAME_WIDTH  = 300;
10    final static int FRAME_HEIGHT = 300;
11
12    public static void main( String args[] )
13    {
14        Border f = new Border();
15        f.setSize( FRAME_WIDTH, FRAME_HEIGHT );
16        //f.pack();                  //  sliver of a window on 68K
17        f.show();
18
19        String [] borders = {
20            "North", "East", "South", "West", "Center"
21        };
22
23        f.setLayout( new BorderLayout( 10, 10 ) );
24
25        for( int i = 0 ; i < borders.length ; i++ )
26        {
27            f.add( borders[i], new Button(borders[i]) );
28        }
29    }
30 }
```

29.3 Card Layout

The card layout is analagous to the use of tabbed dialogs. They are a very efficient way to display a lot of related information. In the curent AWT, this layout is somewhat difficult to use. In JFC 1.0 (Swing), this layout is much easier to use.

The card layout is unique in that objects are laid out *behind each other* instead of next to each other *next to each other*.

29.4 Grid Layout

The grid layout arranges all components in rows and columns (just like a spreadsheet). For a grid layout, cells are always the same size.

The number of rows and columns is specified when constructing a grid layout object.

Components are added starting with the first entry in the first row, then the second entry in the first row, etc.

Grid Layout Configuration

Grid Layout Configuration—After Resizing

```
 1 /*   Grid.java
 2  */
 3
 4 import java.awt.*;
 5 import corejava.*;
 6
 7 public class Grid extends CloseableFrame
 8 {
 9     final static int FRAME_WIDTH  = 300;
10     final static int FRAME_HEIGHT = 300;
11
12     public static void main( String args[] )
13     {
14         Grid f = new Grid();
15         f.setSize( FRAME_WIDTH, FRAME_HEIGHT );
16         f.show();
17
18         f.setLayout( new GridLayout( 0, 3, 10, 10 ) );
19
20         for( int i = 1 ; i <= 9 ; i++ )
21         {
22             f.add( new Button( "Button #" + i ) );
```

```
23            }
24        }
25 }
```

29.5 Grid Bag Layout

The grid bag layout is very flexible, but it can be programmatically tedious to create.

To describe the layout to the grid bag manager, use the following procedure:

- Create an object of type `GridBagLayout`. You don't need to specify how many rows and columns the grid will contain. Java will try use the information specified later.
- Set the `GridBagLayout` object to be the layout manager for the component.
- Create an object of type `GridBagConstraints`. The `GridBagConstraints` object specifies how the components are laid out within the grid bag.
- *For each component*, fill in the `GridBagConstraints` object.
- Add the component with the constraints.

GridBag Layout Configuration

```
 1 /*   GridbagTest.java
 2  *
 3  *   Bruce M. Bolden
 4  *   May 15, 1997
 5  *   Revised:   March 31, 1998
 6  *   http://www.cs.uidaho.edu/~bruceb/
 7  */
 8
 9 import java.awt.*;
10 import java.util.Date;
11 import corejava.*;
12
13
14 public class GridBagTest
15     extends CloseableFrame
16 {
17     static Label       inFileNameLabel  = new Label( "Input File:" );
18     static TextField   inFileName       = new TextField( 45 );
19     static Button      selectInFileNameButton = new Button( "File..." );
20         // File selection dialog box
```

```
21      static FileDialog fileDialog = null;
22
23      static Label      outputDirLabel  = new Label( "Output Directory:" );
24      static TextField  outputDir       = new TextField( 40 );
25      static Button     selectOutputDirButton = new Button( "Directory..." );
26      static FileDialog outputDirDialog = null;
27
28      static Label      processNameLabel = new Label( "Retrieving:" );
29      static TextField  processName      = new TextField( 33 );
30
31      static TextArea  textLog           = new TextArea( 10, 60 );
32
33      static Button     goButton      = new Button( "Go!" );
34      static Button     saveLogButton = new Button( "Save Log file" );
35      static Button     pauseButton   = new Button( "Pause" );
36      static Button     quitButton    = new Button( "Quit" );
37      static Button     hiddenButton  = new Button( "hidden" );
38      static Button     testButton    = new Button( "Test hidden" );
39
40
41      public static void main( String args[] )
42      {
43          final int FRAME_WIDTH  = 500;
44          final int FRAME_HEIGHT = 320;
45
46          GridBagTest f = new GridBagTest();
47
48          f.setTitle( "Fetch 'em" );
49          f.setSize( FRAME_WIDTH, FRAME_HEIGHT );
50          f.initParams();
51          f.initUI();
52          f.show();
53      }
54
55      /**  Initialize input file name and output directory.
56       */
57
58      public void initParams()
59      {
60      }
61
62      /**  Initialize the User Interface
63       */
64
65      public void initUI()
```

```
66      {
67          Panel   p1 = new Panel();
68          Panel   p2 = new Panel();
69          Panel   p3 = new Panel();
70          Panel   p4 = new Panel();
71          Panel   p5 = new Panel();
72
73              // Layout objects for UI
74          FlowLayout     genLayout  = new FlowLayout( FlowLayout.LEFT );
75          GridBagLayout  gbLayout   = new GridBagLayout();
76          GridBagConstraints  gbConstraints  = new GridBagConstraints();
77
78          setLayout( gbLayout );
79              // Left justify most panels
80          gbConstraints.anchor = GridBagConstraints.WEST;
81
82              // Input file name
83          p1.setLayout( genLayout );
84          p1.add( inFileNameLabel );
85          p1.add( inFileName );
86          p1.add( selectInFileNameButton );
87
88              // Output directory
89          p2.setLayout( genLayout );
90          p2.add( outputDirLabel );
91          p2.add( outputDir );
92          p2.add( selectOutputDirButton );
93
94              // Status
95          p3.setLayout( genLayout );
96          p3.add( processNameLabel );
97          p3.add( processName );
98          processName.setEditable( false );
99
100             // Log
101         p4.add( textLog );
102
103             // Control buttons
104         goButton.setBackground( Color.green );
105
106         p5.add( quitButton );
107         p5.add( saveLogButton );
108         p5.add( hiddenButton );
109         p5.add( pauseButton );
110         p5.add( goButton );
```

```
111         p5.add( testButton );
112
113             // Add the panels
114         addComponent( p1, gbLayout, gbConstraints, 0, 0, 2, 1 );
115         addComponent( p2, gbLayout, gbConstraints, 1, 0, 2, 1 );
116         addComponent( p3, gbLayout, gbConstraints, 2, 0, 2, 1 );
117         addComponent( p4, gbLayout, gbConstraints, 3, 0, 2, 1 );
118                 // center last set of buttons
119         gbConstraints.anchor = GridBagConstraints.CENTER;
120         gbConstraints.fill   = GridBagConstraints.HORIZONTAL;
121         addComponent( p5, gbLayout, gbConstraints, 4, 0, 4, 1 );
122
123         hiddenButton.hide();
124         pauseButton.disable();
125
126         inFileNameLabel.requestFocus();
127         Date currentDate = new Date();
128         textLog.appendText( currentDate.toString() + "\n" );
129     }
130
131     /**  Add a component using the GridBag Layout from Deitel & Deitel,
132      *   Java How to Program, 1997.
133      */
134
135     void addComponent( Component c, GridBagLayout gbL,
136                        GridBagConstraints gbC,
137                        int row, int column, int width, int height )
138     {
139         gbC.gridx = column;
140         gbC.gridy = row;
141
142         gbC.gridwidth = width;
143         gbC.gridheight = height;
144
145         gbL.setConstraints( c, gbC );
146
147         add( c );
148     }
149 }
```

29.6 Custom Layout Managers

It is possible to define custom layout managers. They will not be discussed in detail in this class.

30 Applets

An applet is a mini-application, designed to be run using a Web browser or some other *applet viewer*. Applets differ from applications in a number of ways. The most important is that there are security restrictions on what applets are allowed to do when running. Applets often consist of untrusted code, so they cannot be allowed to access the local file system.

From a programming viewpoint, one of the biggest differences between applets and applications is that applets do not have a `main()` method, or other single entry point from which the program starts running.

To write an applet, the `Applet` class is subclassed and a number of standard methods are overridden. At appropriate times, under well-defined circumstances, the Web browser of applet viewer invokes the defined methods. Note: the applet is not in control of the thread of execution—it responds when the browser or viewer tells it to do something.

30.1 Writing an applet

Writing an applet comes down to defining the appropriate methods. A number of these methods are defined by the `Applet` class. The most important methods are: `init()`, `destroy()`, `start()`, `stop()`, `getAppletInfo()`, and `getParameterInfo()`.

`init()`

Called when the applet is first loaded into the browser or viewer. Typically performs applet initialization.

`destroy()`

Called when the applet is about to be unloaded from the browser or viewer.

`start()`
Called when the applet becomes visible and should start doing whatever task it performs. Frequently used with animation and threads.

`stop()`
Called when the applet becomes temporarily invisible, for example, when the user has scrolled it off the screen. Tells the applet to stop performing whatever task it is performing.

`getAppletInfo()`
Called to get information about the applet.

`getParameterInfo()`
Called to get information about the parameters the applet responds to. Should return strings describing those parameters.

30.2 More frequently used methods

The `Applet` class also defines some methods that are commonly used by applets:

`getCodeBase()`
Returns the base URL from which the applet class file was loaded.

`getDocumentBase()`
Returns the base URL of the HTML file that refers to the applet.

`getParameter()`
Looks up and returns the value of a named parameter, specified in the HTML file that refers to the applet with the `<PARAM>` tag.

`getImage()`
Loads an image file from the network and returns an `Image`

object.

`getAudioClip()`
Loads a sound clip from the network and returns an `AudioClip` object.

30.3 Simple Applet

To display an applet, an HTML file containing a reference to it is used.

```
<applet
   code="SimpleApplet.class" width=200 height=100>
</applet>
```

Using a JAR file is slightly more complicated.

```
<applet
   archive="AppletClasses.jar"
   code="SimpleApplet.class" width=200 height=100>
</applet>
```

Either way, the applet may be viewed locally using `appletviewer` using the following command

```
appletviewer SimpleApplet.html
```

The following sample code is about the simplest applet you can write in Java. Note that the `paint()` method is invoked by the applet viewer (or Web browser) when the applet needs to be drawn.

```
/*  SimpleApplet.java
 */
```

```
import java.awt.*;
import java.applet.Applet;

public class SimpleApplet extends Applet
{
   public void init()
   {
      repaint();
   }

   public void paint( Graphics g )
   {
      g.drawString( "Hello World!", 30, 30 );
   }
}
```

Simple Applet

30.4 Event Handling

An applet must receive and respond to user input if it will be interactive. Most browsers do not support the JDK 1.1^{+} event

model that we have been studying.

30.5 Applet using Card Layout

The following applet illustrates the usage of the Card Layout manager. The code is also dependent upon `ImageLabel`, but it it too long to discuss in lecture.

Initial Card Display

Card Display after selecting Ace

```
1 import java.applet.Applet;
2 import java.awt.*;
3
4
5 // This appears in Core Web Programming from
6 // Prentice Hall Publishers, and may be freely used
7 // or adapted. 1997 Marty Hall, hall@apl.jhu.edu.
8
9 /** An example of CardLayout. The right side of the
10  *  window holds a Panel that uses CardLayout to control
11  *  four possible sub-panels (each of which is a
12  *  CardPanel that shows a picture of a playing card).
13  *  The buttons on the left side of the window manipulate
14  *  the "cards" in this layout by calling methods in
15  *  the right-hand panel's layout manager.
16  * @see CardPanel
17  */
18
19 public class CardDemo extends Applet
20 {
21   private Button first, last, previous, next;
22   private String[] cardLabels = { "Jack", "Queen",
23                                   "King", "Ace" };
24   private CardPanel[] cardPanels = new CardPanel[4];
25   private CardLayout layout;
26   private Panel cardDisplayPanel;
27
28
29   public void init() {
30     setBackground(Color.white);
31     setLayout(new BorderLayout());
32     addButtonPanel();
33     addCardDisplayPanel();
34   }
35
36   private void addButtonPanel() {
37     Panel buttonPanel = new Panel();
38     buttonPanel.setLayout(new GridLayout(9, 1));
39     Font buttonFont =
40       new Font("Helvetica", Font.BOLD, 18);
41     buttonPanel.setFont(buttonFont);
42     for(int i=0; i<cardLabels.length; i++)
43       buttonPanel.add(new Button(cardLabels[i]));
44     first = new Button("First");
45     last = new Button("Last");
```

```
46        previous = new Button("Previous");
47        next = new Button("Next");
48        buttonPanel.add(new Label("------------",
49                                  Label.CENTER));
50        buttonPanel.add(first);
51        buttonPanel.add(last);
52        buttonPanel.add(previous);
53        buttonPanel.add(next);
54        add("West", buttonPanel);
55    }
56
57    private void addCardDisplayPanel() {
58        cardDisplayPanel = new Panel();
59        layout = new CardLayout();
60        cardDisplayPanel.setLayout(layout);
61        String cardName;
62        for(int i=0; i<cardLabels.length; i++) {
63          cardName = cardLabels[i];
64          cardPanels[i] = new CardPanel(cardName, getCodeBase(),
65                             "images/" + cardName + ".gif");
66          cardDisplayPanel.add(cardName, cardPanels[i]);
67        }
68        add("Center", cardDisplayPanel);
69    }
70
71    public boolean action(Event event, Object object) {
72        if (event.target == first)
73          layout.first(cardDisplayPanel);
74        else if (event.target == last)
75          layout.last(cardDisplayPanel);
76        else if (event.target == previous)
77          layout.previous(cardDisplayPanel);
78        else if (event.target == next)
79          layout.next(cardDisplayPanel);
80        else
81          layout.show(cardDisplayPanel, (String)object);
82        return(true);
83    }
84 }
```

```
1 import java.awt.*;
2 import java.net.*;
3
4 // This appears in Core Web Programming from
5 // Prentice Hall Publishers, and may be freely used
6 // or adapted. 1997 Marty Hall, hall@apl.jhu.edu.
7
8 /** A Panel that displays a playing card. This window
9  *  does <B>not</B> use CardLayout. Rather, instances
10 *  of CardPanel are contained in another window
11 *  used in the CardDemo example. It is this enclosing
12 *  window that uses CardLayout to manipulate which
13 *  CardPanel it shows.
14 * @see CardDemo
15 */
16
17 public class CardPanel extends Panel {
18   private Label name;
19   private ImageLabel picture;
20
21
22   public CardPanel(String cardName,
23                    URL directory, String imageFile)
24   {
25     setLayout(new BorderLayout());
26     name = new Label(cardName, Label.CENTER);
27     name.setFont(new Font("TimesRoman", Font.BOLD, 50));
28     add("North", name);
29     picture = new ImageLabel(directory, imageFile);
30     Panel picturePanel = new Panel();
31     picturePanel.add(picture);
32     add("Center", picturePanel);
33     resize(preferredSize());
34   }
35
36   public Label getLabel() {
37     return(name);
38   }
39
40   public ImageLabel getImageLabel() {
41     return(picture);
42   }
43 }
```