

1st week discussion

ICS 52: Introduction to Software Engineering

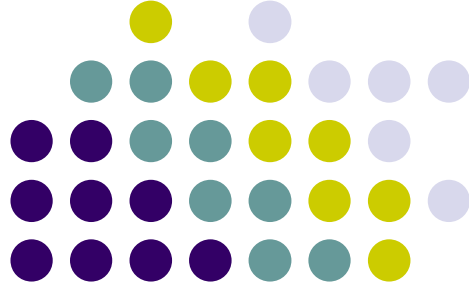
Instructor: Prof. Dan Frost

TA: Derek Pfister

dpfister@uci.edu

Sept. 29, 2008

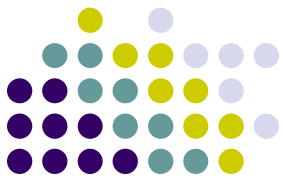
TA OH: 11-12am W F DBH5209





Outline

- History of Java
- New features of Java 1.5
- Assignment 1



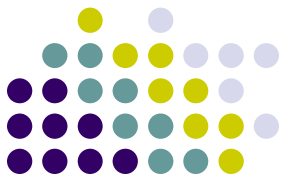
Early days

- Java “NOT” developed for World Wide Web
- Late 70s Bill Joy wanted to rewrite UNIX
 - C++ inadequate for short effective programs
- 1991 – Stealth Project: Smart consumer electronic devices
 - A large, distributed, heterogeneous network of consumer electronic devices all talking to each other
- Development of an independent language
 - Initially named Oak, then renamed to Java



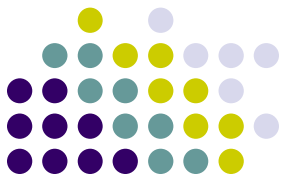
Requirements

- **Platform independence**
 - Compiled vs Interpreted
 - Compromise: VM
- **Reliability:** Crash and reboot not acceptable. Reduce programmer induced errors
 - multiple-inheritance
 - operator overloading
 - implicit garbage collection thereby providing efficient memory utilization and higher reliability
 - eliminate all unsafe constructs used in C and C++ by only providing data structures within objects
- **Security**
 - pointers were excluded



WWW

- Initial efforts of Java not successful
- 1994, WWW came alive
- Java was relaunched as the language of Internet apps
- Sun formally announced Java at SunWorld'95.
- Netscape Inc. announced that it would incorporate Java support in their browser
- Microsoft followed



History of Java

- Java 1.0, start out in 1995, C-like language
 - Automatic garbage collection, applet of web page
- Java 1.1, 1997
 - More scalable event model for GUI programming
 - Introduction of inner classes
- Java 1.2 and 1.3
 - Introduction of Collections framework
 - Incremental improvement of Swing GUI components, other libraries and APIs
- Java 1.4, 2002
 - Introduction of assertions into the core language
 - A logging facility

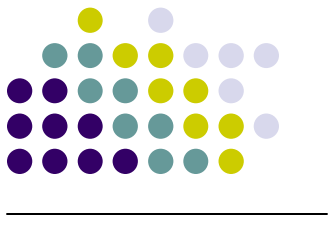
Some New Language Features of Java

1.5

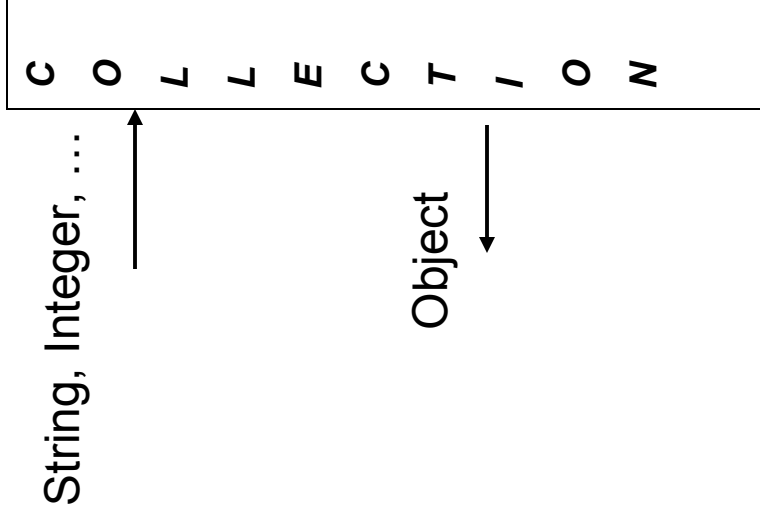
- Generics
- Autoboxing/Unboxing
- Typesafe Enumeration
- Enhanced for loop
- Static import
- Variable Arguments (“Varargs”)



Generics: Raw Collection before Java1.5

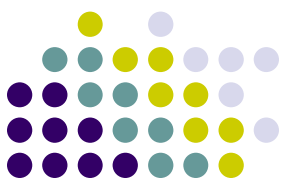


```
.....  
List list = new ArrayList();           // 1  
list.add(new String("Hello world!")); // 2  
list.add(new String("Good bye!"));    // 3  
list.add(new Integer(95));            // 4  
.....  
Iterator i = list.iterator();         // 5  
while(i.hasNext()) {                 // 6  
    String item = (String) i.next();  // 7  
    .....  
}
```



***Explicit cast required in line 7*

*** Program compiles fine, but throws an exception at runtime while trying to cast an Integer to a String (line 7)*

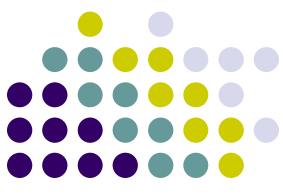


Generics: Generic Collection with Java 1.5

```
.....  
List<String> list = new ArrayList<String>(); // 1  
list.add(new String("Hello world!")); // 2  
list.add(new String("Good bye!")); // 3  
list.add(new Integer(95)); // 4  
.....  
Iterator<String> i = list.iterator(); // 5  
While(i.hasNext()) { // 6  
String item = i.next(); // 7  
.....
```

** Compile time error is produced (line 4)

Generics provide compile-time typesafety for collections and eliminate the need for casts



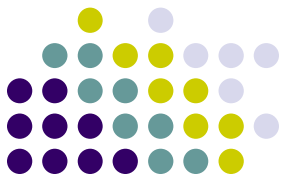
Autoboxing/Unboxing: before Java1.5

Traditional “boxing” example

```
List ssnList = new ArrayList(); // 1
.....
int ssn = getSocSecNum(); // 2
.....
Integer ssnInteger = new Integer(ssn); // 3
ssnList.add(ssnInteger) // 4
```

**** Explicit conversion from int (primitive type) to Integer (wrapper object) - line 3**

**** Can't put primitives into Collections**



Autoboxing/Unboxing: with Java1.5

“Autoboxing” example

```
List ssnList = new ArrayList(); // 1
.....
int ssn = getSocSecNum(); // 2
.....
ssnList.add(ssn) // 3
```

**** No need for an explicit conversion to Integer**

**** Automatic conversion by the compiler**



Autoboxing/Unboxing: with Java1.5

Another “Autoboxing” example

```
double area(double height, double width) { // 1
    return height * width; // 2
} // 3
. .
Double h = new Double (...); // 4
double w = ...; // 5
double a = area(h,w); // 6
Double A = area(h,w); // 7
```

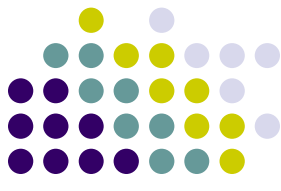
- ** Primitives and Wrapper Objects can be mixed
- ** Automatic unboxing (Wrapper Object -> primitive) in line 6
- ** Automatic boxing (primitive -> Wrapper Object) in line 7



Autoboxing/Unboxing: Caveats

- An Integer expression can have a null value. If your program tries to autounbox null, it will throw a NullPointerException.
- The == operator performs reference identity comparisons on Integer expressions and value equality comparisons on int expressions.
- Finally, there are performance costs associated with boxing and unboxing, even if it is done automatically.

TypeSafe enumerations: before Java 1.5



- Enumerate type is a type whose values consist of a fixed set of constants

```
public class MainMenu {  
    public static final int MENU_FILE = 0;  
    public static final int MENU_EDIT = 1;  
    public static final int MENU_FORMAT = 2;  
    public static final int MENU_VIEW = 3;  
}  
...  
public void handle (int chosenMenu) {  
    .....  
}  
  
MenuManager mm = .....;  
mm.handle(MENU_EDIT);
```

Not Type Safe –

- errors cannot be caught at compile time
- can pass arbitrary values to methods

e.g. mm.handle(52)

Typesafe Enumerations: with Java 1.5



```
public enum MainMenu {FILE, EDIT, FORMAT, VIEW}
```

```
for(MainMenu menu : MainMenu.values()) {  
    switch(menu) {  
        case FILE:  
            System.out.println("FILE Menu");  
            break;  
        case EDIT:  
            System.out.println("EDIT Menu");  
            break;  
        case FORMAT:  
            System.out.println("FORMAT Menu");  
            break;  
        case VIEW:  
            System.out.println("VIEW Menu");  
            break; } } }
```

- Provides strong compile-time type safety
- Provides several members e.g. `values()`
- * ***enum*** is reserved

Enhanced For Loop: before Java 1.5



Example A

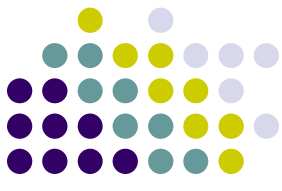
```
List policies = new ArrayList();  
...  
Iterator iter = policies.iterator();  
while ( iter.hasNext() ) {  
    renew( (Policy)iter.next() );  
}
```

Example B

```
List policies = new ArrayList();  
...  
for (Iterator iter =  
    policies.iterator();  
        iter.hasNext(); ) {  
    renew( (Policy)iter.next() );  
}
```

1. *Declare an iterator*
2. *Explicitly check whether another object is available as a loop condition*
3. *Retrieve the object if there is one available*

Enhanced For Loop: with Java 1.5

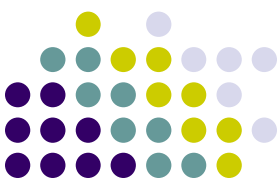


```
List policies = new ArrayList(); // 1
...
for ( Object policy : policies ) { // 2, Object returned
    renew( (Policy)policy ); // 3, Explicit conversion
} // 4
```

With Generic Types

```
List<Policy> policies = new ArrayList<Policy>();
...
for ( Policy policy : policies ) {
    renew( policy );
}
```

Enhanced For Loop



- *Iterating over an array : before Java 1.5*

```
public int sumArray(int array[]) { // 1
    int sum = 0; // 2
    for(int i=0;i<array.length;i++) { // 3
        sum += array[i]; // i -> index // 4
    }
    return sum; } //6
```

- *Iterating over an array : with Java 1.5*

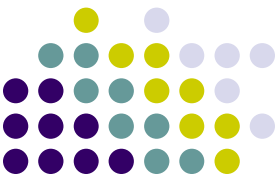
```
public int sumArray(int array[]) {
    int sum = 0;
    for(int i : array) {
        sum += i; // i -> array element
    }
    return sum; }
```

- **Enhanced for loop** cannot be used to remove or modify elements in a collection

Static Import: before Java 1.5

```
import java.lang.Math.*;
import java.lang.System.*;
...
double areaCircle = Math.PI * Math.pow(10.0, 2.0);
...
System.out.println("Hello World");
```

** Importing a class or interface requires a class name prefix



Static Import: with Java 1.5



```
import static java.lang.Math.*; // 1
import static java.lang.System.*; // 2
....
double areaCircle = PI * pow(10.0, 2.0); // 3
...
out.println("Hello World"); // 4
```

** No need to reference Math or System except in
import



Variable Length Argument: before Java 1.5

- How do you pass a variable number of arguments to a method?.....Use an array

```
double sum(double[] dArg) {  
    double total = 0.0;  
    for ( double d : dArg )  
        total += d;  
    return total;  
}
```

.....but you have to load the array

```
double dblArray[] = { . . . };  
  
double total = sum(dblArray);
```

Variable Length Argument: **with** Java 1.5



```
double sum(double ... dArg) { // 1
    double total = 0.0; // 2
    for ( double d : dArg ) // 3
        total += d; // 4
    return total; // 5
}
```

- The notation (...) allows a variable number of arguments
- Arguments to the method treated like an array

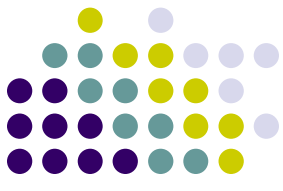
```
.....
sum(1.1, 2.2);
sum(1, 2.3, 4.56, 7.89, 101);
sum();
.....
```



Assignment1: Enhance a mini library system

- What you have: LibFiles.zip
 - Book.java
 - BookCollection.java
 - Cardholder.java
 - ChList.java
 - EnterISBNDialog.java
 - Lib.java
 - LibFrame.java
 - Titles.dat

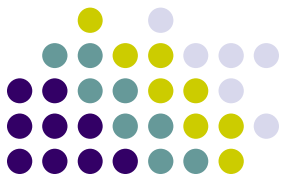
Assignment1: Enhance a mini library system



- What you need do:
 - Fix warnings: `javac -Xlint *.java`
 - Implement search by Title, Author and Keyword
(Note: using Iterators defined in `BookCollection.java`)
 - Implement four functionalities:
 - More flexible ISBN search by ignoring dashes
 - Add exit and save function
 - Make checked out books to be checked back in
 - Permit user to reserve a book (`book.java`)

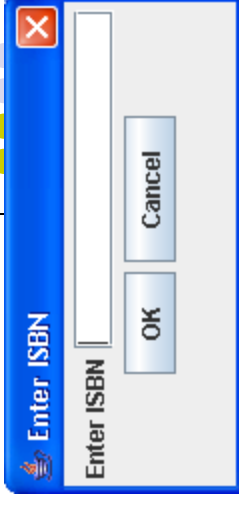
JFrame

- It's a window with title, border, (optional) menu bar and user-specified components.
- It can be moved, resized, minimized.
- *The LibFrame class in the assignment extends JFrame*



JDialog [EnterISBNDialog]

- Like a frame, a dialog box is a separate window.
- Unlike a frame, however, a dialog box is not completely independent.
- Every dialog box is associated with a frame, which is called its parent.



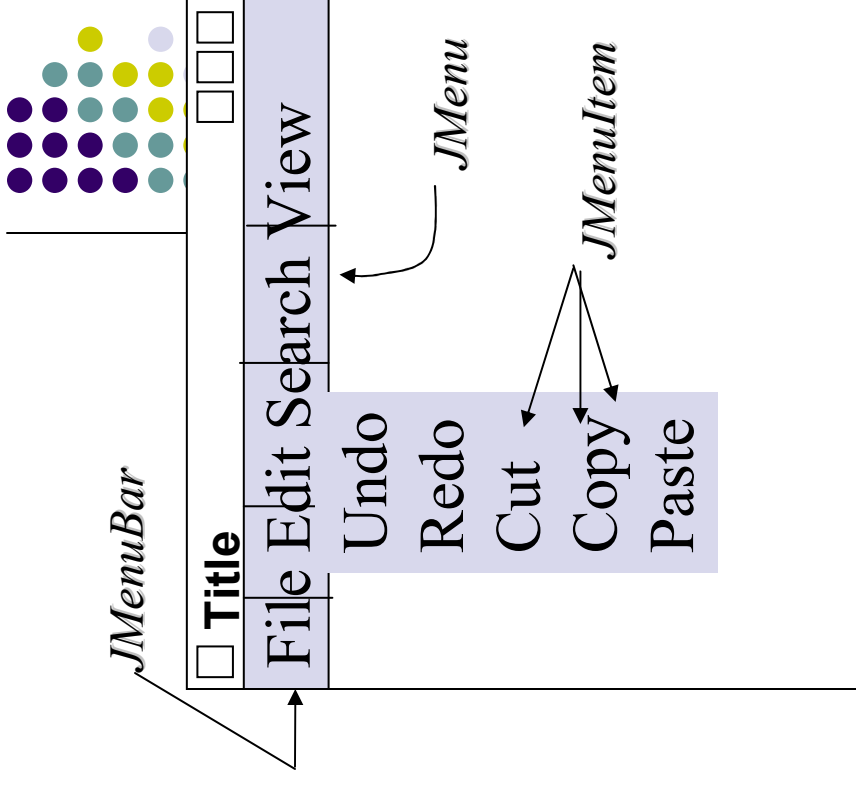
```
cancelButton.addActionListener(buttonListener); // 1
okButton.addActionListener(buttonListener);   // 2
.....
Object object = event.getSource();           // 3
if (object == okButton)                      //4
{
.....
```

- **Create dialog boxes for the other box searches (title, author, keyword)**

JMenuBar

```
public class MenuFrame extends JFrame // 1
implements ActionListener // 2
{
    public MenuFrame() {
        // Set up frame itself – title,size,location

        JMenuBar menuBar = new JMenuBar( ); // 7
        setJMenuBar( menuBar ); // 8
        JMenu fileMenu = new JMenu( "File" ); // 9
        menuBar.add( fileMenu ); // 10
        JMenu editMenu = new JMenu( "Edit" ); // 11
        .....
        JMenuItem item; // 12
        item = new JMenuItem ( "Undo" ); // 13
        item.addActionListener( this ); // 14
        editMenu.add( item ); // 15
        item = new JMenuItem ( "Redo" ); // 16
        item.addActionListener( this ); // 17
```



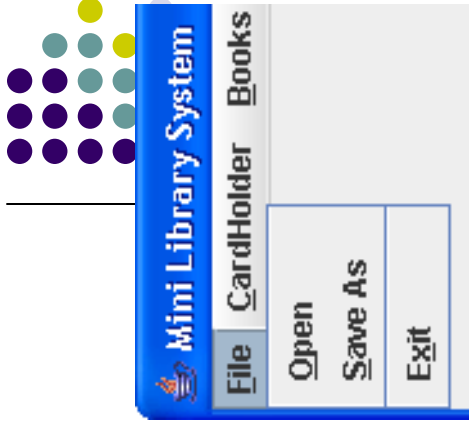
- Create a menu bar and add menus to it (lines 7, 10)
- Add the menu bar to the frame (8)
- Create menu objects and add items to them (lines 11, 15)
- Create menu items objects for your actions (lines 13, 16)

JMenuBar [LibMenuBar]

```
JMenu fileMenu, chMenu, bkMenu; // 1
JMenuItem menuItem; // 2
// Build the File menu.
fileMenu = new JMenu("File"); // 4
fileMenu.setMnemonic(KeyEvent.VK_F); // 5
add(fileMenu); // 6

// attach to it a group of JMenuItems
menuItem = new JMenuItem("Open", KeyEvent.VK_O); // 8
menuItem.addActionListener(new FileOpenListener()); // 9
fileMenu.add(menuItem); // 10
.....

class FileOpenListener implements ActionListener { // 11
    public void actionPerformed(ActionEvent e) { // 12
        parentFrame.loadData(); // 13
    } // 14
}
```



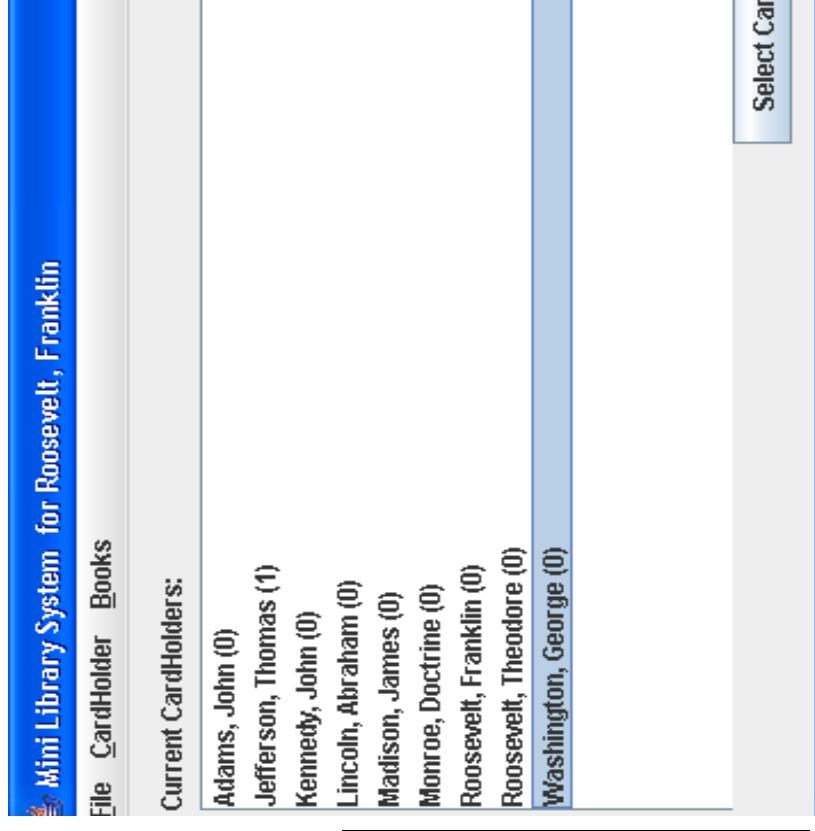
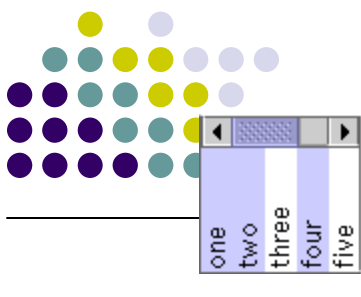
JList

- Allows users to select one or more items from a list
- Each **JList** has a

ListSelectionModel

- keeps track of selected items
- enforces selection mode (single, single interval, multiple interval)
- notifies listeners of selection changes

```
DefaultListModel chListModel = new DefaultListModel(); // 1
chList = new JList(chListModel); // 2
chList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); // 3
chList.setSelectedIndex(0); // 4
```



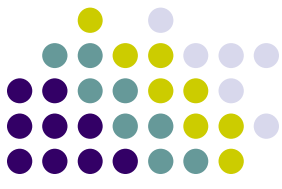
**** SelectCardHolder() in LibFrame might come in useful for other book searches.**



Main Steps

- To make any graphic program work we must be able to create windows and add content to them.
- To make this happen we must:
 1. Import the swing packages.
 2. Set up a top-level container.
 3. Fill the container with GUI components.
 4. Install listeners for GUI Components.
 5. Display the container.

Serialization



- *Object serialization* provides the ability to save an object, and its current state, so that it can be reused in another program
- The idea that an object can “live” beyond the program execution that created it is called ***persistence***
- Object serialization is accomplished using the `Serializable` interface and the `ObjectOutputStream` and `ObjectInputStream` classes
- The `writeObject` method is used to serialize an object
- The `readObject` method is used to deserialize an object

Serialization

From LibFrame.java

```
public void saveData()  
.....  
FileOutputStream fileOut = new FileOutputStream(file);  
ObjectOutputStream out = new ObjectOutputStream(fileOut);  
out.writeObject(lib.getChList()); // serialize  
out.writeObject(lib.getBookCollection()); // serialize
```



Serialization

From LibFrame.java

```
public void loadData()  
.....  
FileInputStream fileIn = new FileInputStream(file);  
ObjectInputStream in = new ObjectInputStream(fileIn);  
lib.loadChList((ChList)in.readObject());  
lib.loadBookCollection((BookCollection)in.readObject());
```

