Object Oriented Programming

1. Applets
2. Java Collections

What's a Java Applet?

- Applets are simply java programs that run in web browsers.
- Created a big stir in mid-90's when Netscape agreed to embed JVM in Navigator web browser.
- Great promise – applications automatically distributed to anyone on any platform!
- Reality – non-uniform browser support, limitations imposed by security, easier ways to accomplish same thing!

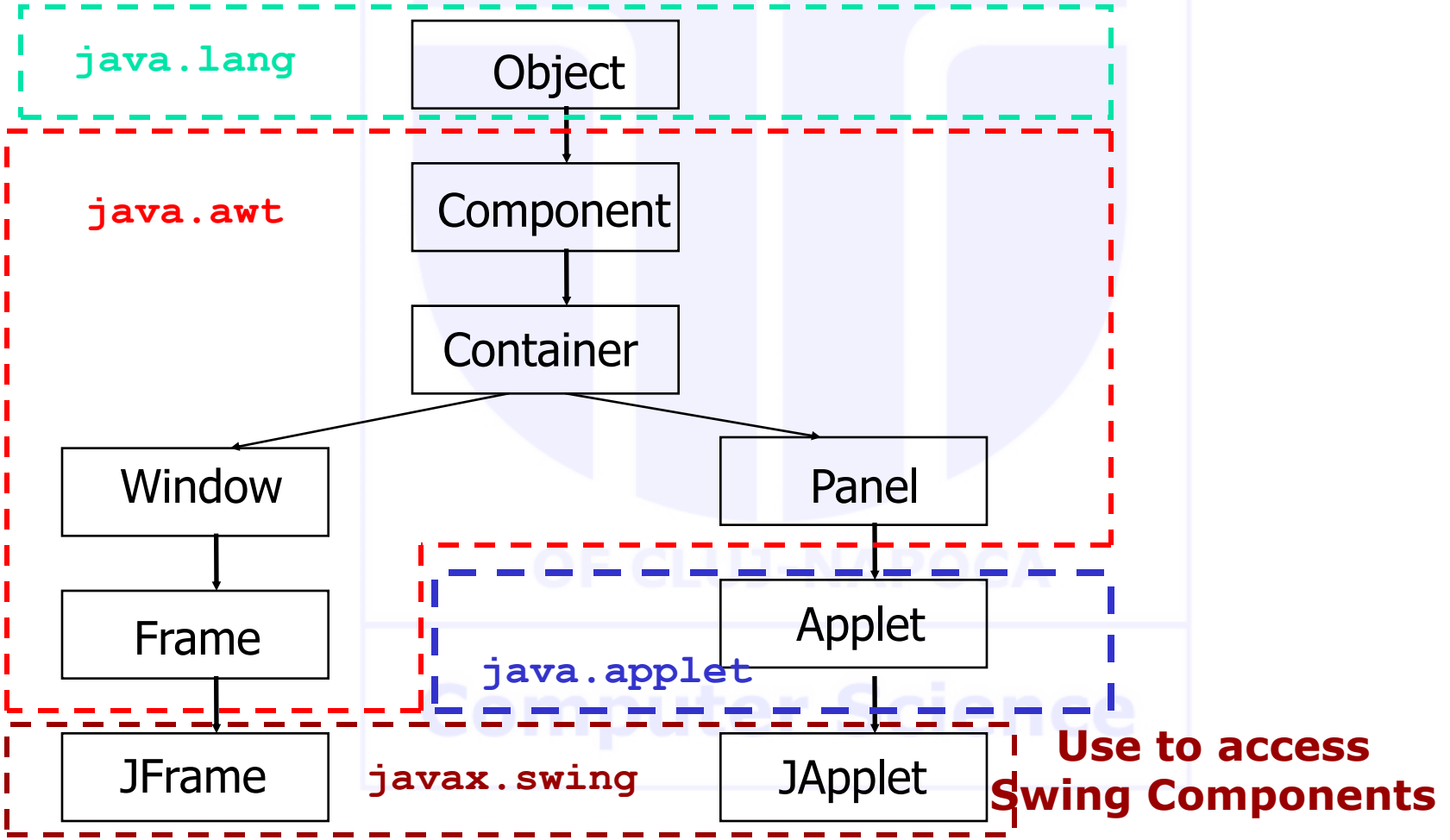
What's a Java Applet?

- Still useful in just the right situation
  - fancy, full-fledged client
  - can make some assumptions/have some control over client technology
- Also, very good for understanding issues in web client-server programming
- Otherwise, server-heavy programming with HTML or client-side scripting (using, e.g. JavaScript) wins out.
- Also, Java WebStart – new alternative
  - It enables applications to be downloaded from within a browser, but then run independently of the browser, using the system's own Java Virtual Machine.

Practical Uses of Applets

- Applets have serious uses on intranets. This is for a number of reasons.
  - Corporate intranets are secure and managed -> many of the normal security restrictions can be relaxed.
  - Ease of management. Large numbers of very small applications distributed over large numbers of machines need to be managed (e.g. periodically updated) and less than capable users can move, damage or delete them. Applets on a central server downloaded as needed do not have that problem (though it is necessary to maintain appropriate browsers, plug-ins etc.).
  - High bandwidth. A corporate intranet will generally have a relatively high bandwidth - repeatedly downloading even quite large applets may not be an issue.
- Still represents a viable technology for applications that require a more robust interface than that of an HTML page with JavaScript.

Applet inheritance tree



Primary Differences between Applets and Standalone Applications

- An instance of the applet is always created, and its constructor, `init`, and `start` methods are always run.
- Because an `Applet` instance is a `Panel` instance (and `JApplet` inherits from `Applet`), a visible component is created, `awt` events are (potentially) handled, etc.
- When a standalone application is invoked, ***only*** `public static void main( String[] )` (and code called by it) is run.

Applet Methods

- Every applet has 5 standard methods:
  - `init()` — called by the browser when the applet is first loaded into memory
  - `start()` — called by the browser to start animations running when the applet is made visible
  - `stop()` — called by the browser to stop animations running when the applet is covered or minimized
  - `destroy()` — called by the browser just before the applet is destroyed
  - `paintComponent()` — called when the applet is drawn or re-drawn

Applet Methods

- Applet methods will always be called in the order **`init`**, **`start`**, **`stop`**, **`destroy`** – applet's life cycle
- **`start`** and **`stop`** may be called many times during the life of an applet
- All 5 methods are implemented as dummy methods in class **`JApplet`**, and the dummy methods are *inherited* by all applets
- Applets override *only* the methods that they need to perform their function
- An applet has a status window
  - Informs you of what the applet is currently doing
  - To output to the status window use **`showStatus`** method with a **`String`** argument

Creating an Applet

- To create an Swing-based applet :
  - Create a subclass of `JApplet` to hold GUI components
  - Select a layout manager for the container, if the default layout manager (`BorderLayout;` default layout is `FlowLayout` for `Applet`) is not acceptable
  - Create components and add them to *the content pane* of the `JApplet` container.
  - Create "listener" objects to detect and respond to the events expected by each GUI component, and assign the listeners to appropriate components.
  - Create an HTML text file to specify to the browser which Java applet should be loaded and executed

Example: Creating an Applet

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FirstApplet extends JApplet {
  // Instance variables
  private int count = 0;            // Number of pushes
  private JButton pushButton; // Push button
  private JLabel label;             // Label
  // Initialization method
  public void init() {
     // Set the layout manager
    getContentPane().setLayout( new BorderLayout() );
    // Create a label to hold push count
    label = new JLabel("Push Count: 0");
    getContentPane().add( label, BorderLayout.NORTH );
    label.setHorizontalAlignment( label.CENTER );
    // Create a button
    pushButton = new JButton("Test Button");
    pushButton.addActionListener( new ButtonHandler(this) );
    getContentPane().add( pushButton, BorderLayout.SOUTH );
  }
 // Method to update push count
  public void updateLabel() {   label.setText( "Push Count: " + (++count) );   }
}
```

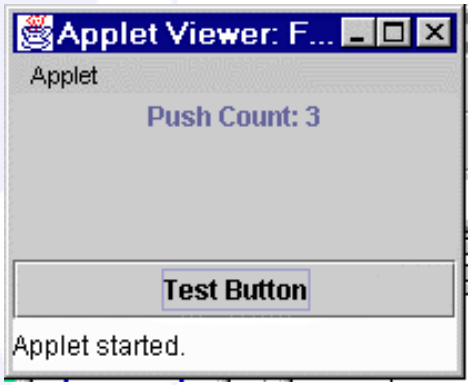
This simple applet implements `init` and *inherits* all other methods as dummies

Note that all components are added to the `ContentPane` of the `JApplet`

Example: Creating an Applet

```
class ButtonHandler implements ActionListener
{
  private FirstApplet fa;
 // Constructor
  public ButtonHandler ( FirstApplet fa1 )
 {
    fa = fa1;
  }
  // Execute when an event occurs
  public void actionPerformed( ActionEvent e )
 {
    fa.updateLabel();
  }
}
```

Listener class for button on applet




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

HTML code to start applet in browser

Applet HTML tags

```
<APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = widthInPixels HEIGHT = heightInPixels
[ALIGN = alignment]
[VSPACE = vspaceInPixels] [HSPACE = hspaceInPixels]
[<PARAM NAME = parameterName1 VALUE = parameterValue1>]
[<PARAM NAME = parameterName2 VALUE = parameterValue2>]
...
[<PARAM NAME = parameterNamen VALUE = parameterValuen>]
>
[HTML that will be displayed in the absence of Java]
</APPLET>
```

Parameters to Applets

- **Parameters are:**
  - **stored as part of the Web page that contains an applet.**
  - **created using the HTML tag <PARAM> and its two attributes: NAME and VALUE.**
- **You can have more than one <PARAM> tag with an applet, but all of them must be between the opening <APPLET> tag and the closing </APPLET> tag. An example with several parameters:**

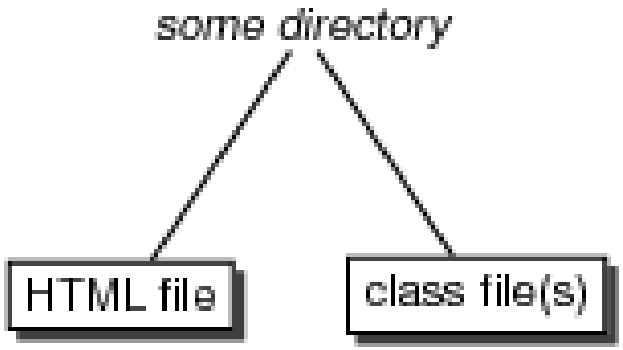
```
<APPLET CODE="ScrollingHeadline.class" HEIGHT=50 WIDTH=400>
<PARAM NAME="Headline1" VALUE="Highest marks assigned">
<PARAM NAME="Headline2" VALUE="All people took part">
</APPLET>
```

- **Use the NAME attribute to give the parameter a name. The VALUE attribute gives the named parameter a value.**
- **Getting parameters: use method getParameter. E.g.**

```
String display1 = getParameter("Headline1");
```

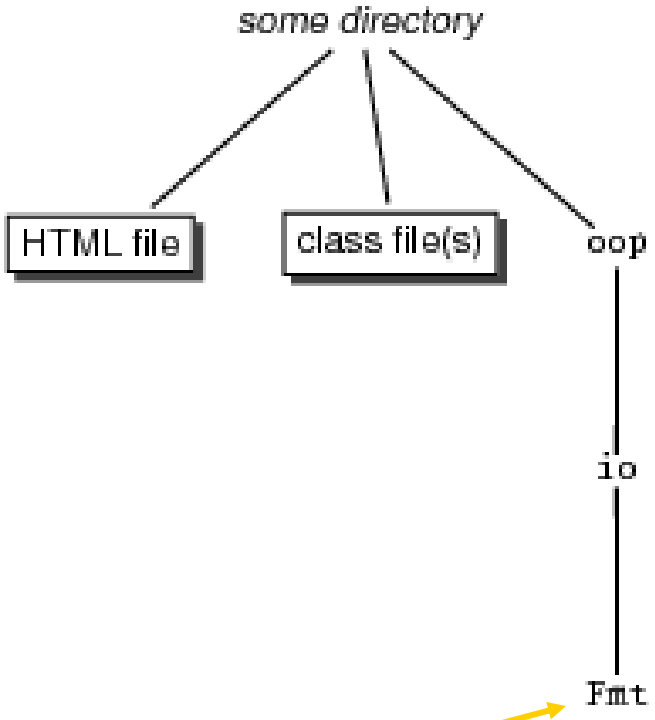
Location of Class Files for Applets

- If an applet uses a class that is *not* built into a package, that class must be present in the *same directory* as the HTML file used to start the applet
  - This directory could be local or remote—it doesn't matter
  - Class files can be transferred over the net, so they should be small



Using Packages with Applets

- If an applet uses *non-standard* package, then *the package must appear in the appropriate subdirectory* of the directory containing the HTML file.
  - This directory could be local or remote—it doesn't matter
  - **`CLASSPATH`** is *ignored* by applets!



Location of class **`oop.io.Fmt`** within package **`oop.io`**

Creating Dual Application / Applets

- If an application does not need to perform I/O or other restricted tasks, it can be structured to run both as an applet and an application
  - Design the program as an applet
  - Add a `main` method with calls to `init` and `start`
  - Add calls to `stop` and `destroy` in the `windowClosing` method of the `Window` handler
- Such a program can be more versatile

Dual Application / Applet

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import chapman.io.*;
public class TempConversionApplet extends JApplet {
   // Instance variables
   private JLabel l1, l2;          // Labels
   private JTextField t1, t2;      // Text Fields
   private DegCHandler cHnd;       // ActionEvent handler
   private DegFHandler fHnd;       // ActionEvent handler
   // Initialization method
   public void init() {

      ...
   }
```

`init` method in applet

Dual Application / Applet

```
    // Main method to create frame
    public static void main(String s[]) {
        // Create a frame to hold the application
        JFrame fr = new JFrame("TempConversionApplet ...");
        fr.setSize(250,100);
        // Create and initialize a TempConversionApplet object
        TempConversionApplet tf = new TempConversionApplet();
        tf.init();
        tf.start();
        // Create a Window Listener to handle "close" events
        AppletWindowHandler l = new AppletWindowHandler(tf);
        fr.addWindowListener(l);
        // Add the object to the center of the frame
        fr.getContentPane().add(tf, BorderLayout.CENTER);
        // Display the frame
        fr.setVisible( true );
    }
}
```

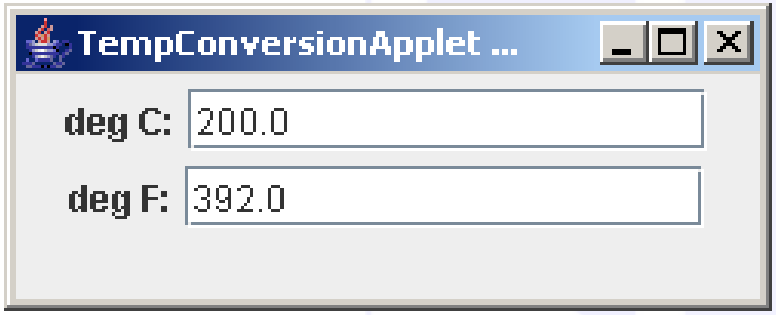
`main` method calls `init` and `start`

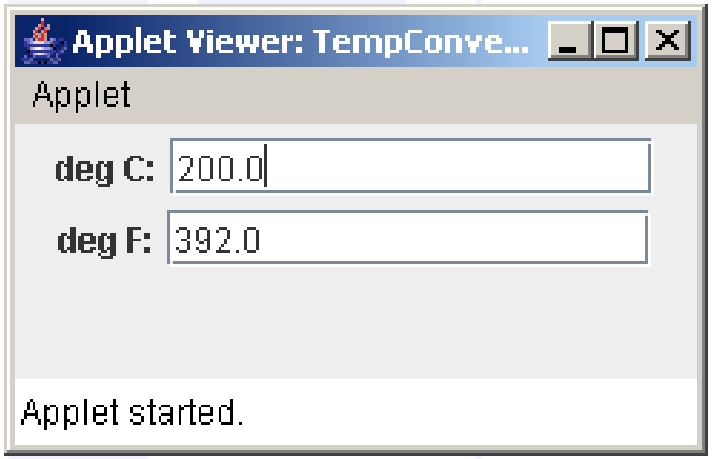
Dual Application / Applet

```
public class AppletWindowHandler extends WindowAdapter {
    JApplet ap;
    // Constructor
    public AppletWindowHandler ( JApplet a ) { ap = a; }
    // This method implements a listener that detects
    // the "window closing event", shuts down the applet,
    // and stops the program.
    public void windowClosing(WindowEvent e) {
        ap.stop();
        ap.destroy();
        System.exit(0);
    };
}
```

`windowClosing` method calls `stop` and `destroy`

Dual Application / Applet





Running as application

Running as applet

Tip: Converting an Existing Application to an Applet (generic guide)

- Create the HTML that will invoke the applet.
- The top-level class should extend **`JApplet`** and not **`JFrame`**. Give the **`JApplet`** the **`BorderLayout`** that the **`JFrame`** uses by default.
- Replace the class constructor with an **`init()`** method to perform the applet setup.
- Move any initialization statements from **`main()`** into **`init()`.** Other statements that do a 'revisit page' setup must be placed in a method called **`start()`.**
- Modify/remove any code that may not be used within an applet, such as setting the title bar, calling native routines, I/O, OS commands, etc.
- Place any clear-up or stop code in the **`stop()`** method.

Tip: Applet to Application Conversion

- Make the top-level class extend **`JFrame`** and not **`JApplet`**.
- Add a **`main`** routine to the class. Care must be taken here to ensure that the **`main`** routine might have to pass on arguments in the same way that parameters may be passed to an applet via HTML.
- The **`main`** routine should create an instance of the class. In turn, the constructor of the class should call (or contain) the **`start()`** and **`init()`** methods.
- Add a menu with exit item/exit button, to ensure a means of exiting the application is provided.

Using Swing in Applets

- JApplet is a *Top-Level Container*
- Some Web Browsers don't support the running of Applets that have Swing Components
- In such cases, the necessary *Plug-In* has to be installed
- *Applet Viewer* supports Swing

```
import javax.swing.*;
public class TestJApplet extends JApplet
{
   public void init()
   {
      // Get the content pane
      Container container = getContentPane();
      JButton button = new JButton("Button");
      container.add(button); // Add it to the content pane
   }
}
```

JApplets vs Regular Applets

- Components are added to Swing applet's *content pane* not directly to the applet
- Layout managers are set to Swing applet's content page not directly to the applet
- Default layout manager is **`BorderLayout`** (not **`FlowLayout`** as regular applets)
- Supports assistive technologies (for people with disabilities)
- Supports adding menu bars

Applet Limitations

- Applets run inside a "sandbox"
- Applet security manager protects the user
- Applets can NOT:
  - Access files from the client machine.
  - Create files on the client machine.
  - Make network connects except back to the host machine where the applet came from.
  - Start any programs on a client machine.
  - Load libraries.
  - Define native method calls.
  - Halt execution of the interpreter (can't call `System.exit()`).

More Applet Restriction Details

- Can never run any client executable
- Cannot communicate with any host other than the server from which it came, "originating host"
- Cannot read or write to the client's file system
- Can find out only limited info about the client machine:
  - Java version in use
  - Name and version of OS running
  - Characters used as file and line separators
  - Client language (English) & locale (Eastern Europe)
  - Client currency (Euro)
- Windows popped up by an applet carry a warning message

Java Collections

Limitations of Arrays

- When managing series, sets, and groups of data arrays are not always the best solution
- Arrays do not excel when data is volatile – especially when the size of the data set can fluctuate
  - insertion of an element requires sliding elements above the insertion point -> need extra space allocated at the end
- More generally, arrays expose low-level memory management issues to the application programmer
- If one wants to provide access to a private array:
  - can make the array itself available through an accessor method
  - can provide an interface for iteration over the array with methods: first, next, etc.
  - can return a deep copy of the array – very inefficient

Collections vs Arrays

- Working with Collections API is different than working with arrays
- Arrays are *strongly typed*
  - You specify the types of elements and the compiler enforces that when you try to assign values to elements
  - You can define arrays of primitive type elements
- Collections are *weakly typed*
  - There is one `Vector` class and all its elements are of type `Object` -> all objects of all types can live there and you must downcast elements when you read them
  - You cannot include primitive values in collections although there is a way to box them

Collections vs Arrays

- Arrays are generally faster as they represent direct, random access to blocks of memory
- Collections are objects with methods that have to be called to read or write elements
- Collections offer some general advantages:
  - They are much easier to use for general purpose coding, especially when data is highly volatile – lots of adds, deletes, and direct changes over time
  - The iterator design split helps hide implementation choices

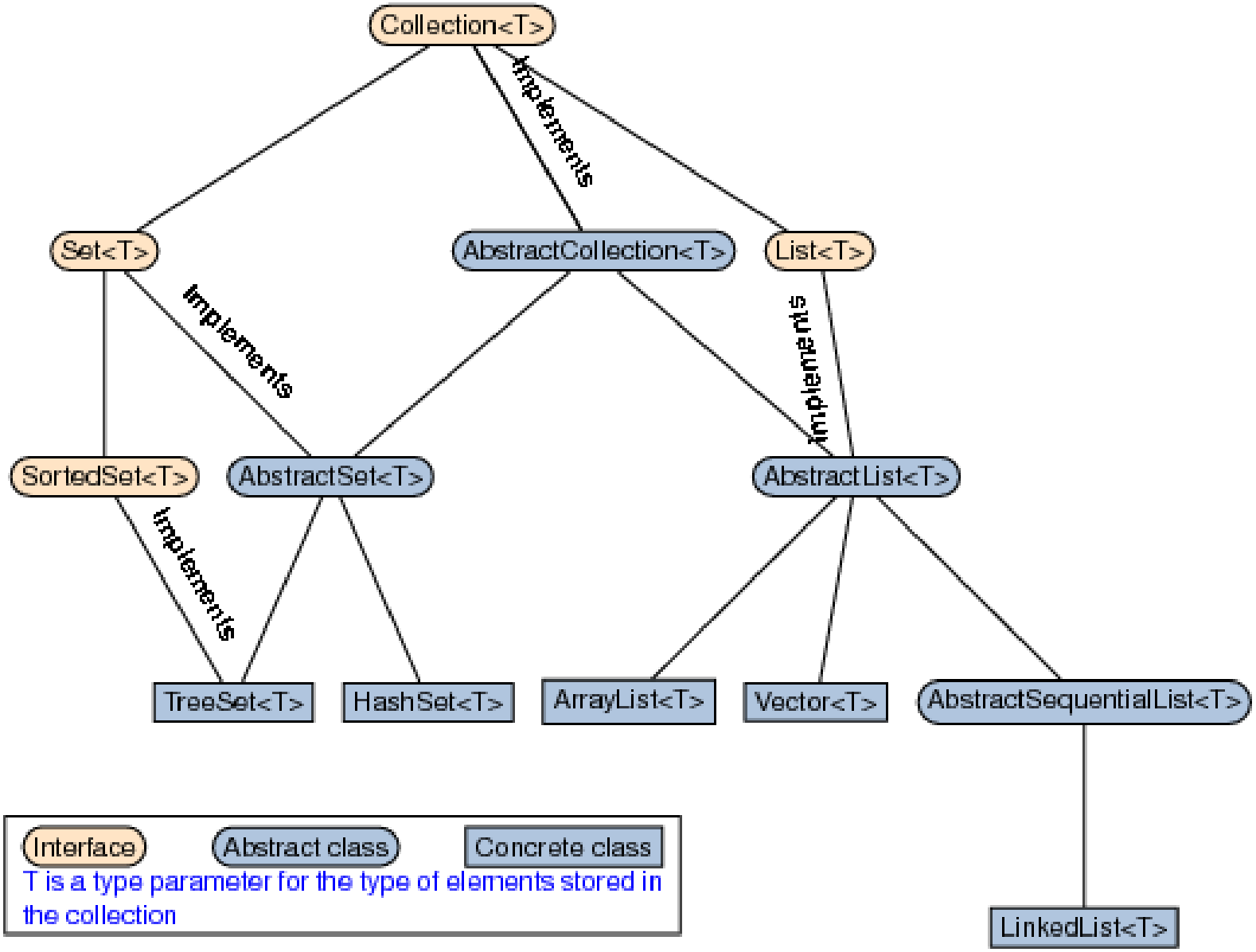
Collections

- *A Java collection* : any class that holds objects and implements the **`Collection`** interface
  - For example, the **`ArrayList<T>`** class is a Java collection class, and implements all the methods in the **`Collection`** interface
  - Collections are used along with *iterators*
- The **`Collection`** interface is the highest level of Java's framework for collection classes
  - All of the collection classes discussed here can be found in package **`java.util`**

Collections

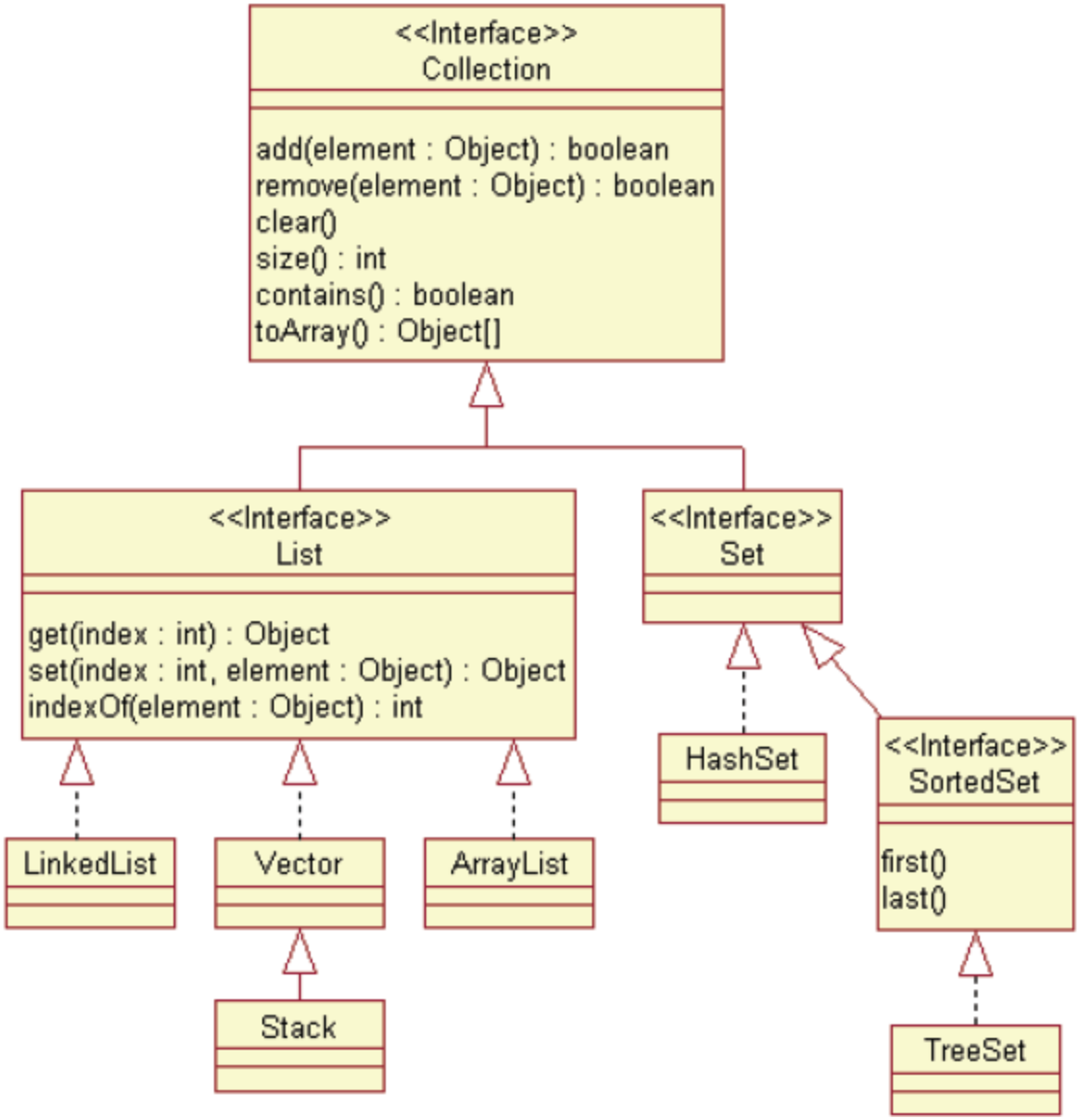
- The collection API includes:
  - Collections such as **`Vector`**, **`LinkedList`**, and **`Stack`**
  - *Maps* that index values under keys, such as **`HashMap`**
  - Variants the assure that elements are always ordered by a comparator: **`TreeSet`** and **`TreeMap`**
  - *Iterators* that abstract the ability to read and write the contents of collection in loops, and isolate that ability from the underlying collection implementation

The `Collection` Landscape



The `Collection` Landscape

- Ordered collections implement **`List`**
- Collections that assure element uniqueness implement **`Set`**
- Sorted collections implement **`SortedSet`**



Wildcards

- Classes and interfaces in the collection framework can have parameter type specifications that do not fully specify the type plugged in for the type parameter
  - Because they specify a wide range of argument types, they are known as *wildcards*

```
public void method(String arg1,
  ArrayList<?> arg2)
```

  - In the above example, the first argument is of type **`String`**, while the second argument can be an **`ArrayList<T>`** with any base type

Wildcards

- A bound can be placed on a wildcard specifying that the type used must be an ancestor type or descendent type of some class or interface
  - The notation `<? extends String>` specifies that the argument plugged in be an object of any descendent class of `String`
  - The notation `<? super String>` specifies that the argument plugged in be an object of any ancestor class of `String`

The `Collection` Framework

- The `Collection<T>` interface describes the basic operations that all collection classes should implement
- Since an interface is a type, any method can be defined with a parameter of type `Collection<T>`
  - That parameter can be filled with an argument that is an object of any class in the collection framework

The `Collection` Interface

- All collections can:
  - add / remove elements
  - clear to an empty set
  - report their size
  - convert data into an array of `Object`s

```
interface Collection
  { // partial listing of methods
    public int size();
    public void clear();
    public Object[] toArray();
    public boolean add( Object );
    public boolean remove( Object );
    public boolean addAll( Collection );
    public Iterator iterator()
```

- Additional properties defined by implementation of one of the sub-interfaces of `Collection`
  - ordered collections implement `List`
  - collections that ensure element uniqueness implement `Set`
  - collections that sort implement `SortedSet`

Building Collections

- You must *create* collections explicitly
  - Common mistake: declare a reference to a `Vector` or `LinkedList` and just assume that the object is there
- Once a collection object is created, simply add elements to it
  - use `add` to append a new element at the end. Note that primitive values must be boxed. E.g.

```
vec.add(new Integer(5))
int i = ((Integer) vec.elementAt(0)).intValue();
Boolean b = ((Boolean)
  vec.elementAt(2)).booleanValue();
```

  - use insert methods – defined by subtypes of `Collection`
  - `remove` an element by identifying it. Many subtypes offer index-based remove methods
- Any Java object can be placed in any collection
  - homogeneous vs heterogeneous collections

Pitfall: Optional Operations

- When an interface lists a method as "optional," it must still be implemented in a class that implements the interface
  - The optional part means that it is permitted to write a method that does not completely implement its intended semantics
  - However, if a trivial implementation is given, then the method body should throw an **`UnsupportedOperationException`**

Tip: Dealing with All Those Exceptions

- The tables of methods for the various collection interfaces and classes indicate that certain exceptions are thrown
  - These are unchecked exceptions, so they are useful for debugging, but need not be declared or caught
- In an existing collection class, they can be viewed as run-time error messages
- In a derived class of some other collection class, most or all of them will be inherited
- In a collection class defined from scratch, if it is to implement a collection interface, then it should throw the exceptions that are specified in the interface

Concrete Collections Classes

- The `ArrayList<T>` and `Vector<T>` classes implement the `List<T>` interface, and can be used directly if additional methods are not needed
  - Both the `ArrayList<T>` and `Vector<T>` classes implement all the methods in the interface `List<T>`
  - Either class can be used when a `List<T>` with efficient random access to elements is needed
- The concrete class `HashSet<T>` implements the `Set<T>` interface, and can be used if additional methods are not needed
  - The `HashSet<T>` class implements all the methods in the `Set<T>` interface, and adds only constructors
  - The `HashSet<T>` class is implemented using a *hash table*

The `Vector` Class

- Provides random access to a scalar list of elements
  - `Vector` and `ArrayList` have semantics most like a raw array

```
for (int n = 0; n < vec.size(); n++)
   System.out.println((String) vec.elementAt(n));
```

  - Elements are ordered based on how they were added to the sollection – there is no implicit sorting
  - Elements need not be unique within the collection
- Vectors perform best in "random access" to their elements – they're backed by arrays
  - weakness – as with arrays – insertion and deletion
- Vectors have capacity and size
  - `size` = number of elements currently in the collection
  - capacity = current allocation of "slots" for elements;
  - capacity ≥ `size`

The Class `ArrayList`

- Creating:
  - `new ArrayList()`
  - `new ArrayList(int initialCapacity)`
- Measuring:
  - `int size()`
- Storing:
  - `boolean add(Object o)`
  - `boolean add(int index, Object element)`
  - `Object set(int index, Object element)`
- Retrieving:
  - `Object get(int index)`
  - `Object remove(int index)`
- Testing:
  - `boolean isEmpty()`
  - `boolean contains(Object elem)`
- Finding (failure = -1):
  - `int indexOf(Object elem)`
  - `int lastIndexOf(Object elem)`

Differences Between `ArrayList<T>` and `Vector<T>`

- For most purposes, the `ArrayList<T>` and `Vector<T>` are equivalent
  - The `Vector<T>` class is older, and had to be retrofitted with extra method names to make it fit into the collection framework
  - The `ArrayList<T>` class is newer, and was created as part of the Java collection framework
  - The `ArrayList<T>` class is supposedly more efficient than the `Vector<T>` class also

Example: `ArrayList`

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorReferenceDemo
{
   public static void main(String[] args)
   {
      ArrayList<Date> birthdays = new
     ArrayList<Date>( );

      birthdays.add(new Date(1, 1, 1990));
      birthdays.add(new Date(2, 2, 1990));
      birthdays.add(new Date(3, 3, 1990));

      System.out.println("The list contains:");

      Iterator<Date> i = birthdays.iterator( );
      while (i.hasNext( ))
         System.out.println(i.next( ));
      i = birthdays.iterator( );

      Date d = null; //To keep the compiler happy.
      System.out.println("Changing the
     references.");
      while (i.hasNext( ))
      {
         d = i.next( );
         d.setDate(4, 1, 1990);
      }

      System.out.println("The list now
     contains:");

      i = birthdays.iterator( );
      while (i.hasNext( ))
         System.out.println(i.next( ));

      System.out.println("April fool!");
   }
}
```

Example: Finding Duplicate Strings

```
import java.util.*;
public class FindDups {
   public static void main(String args[]) {
      Set<String> s = new HashSet<String>();
      for (String a : args)
        if (!s.add(a))
          System.out.println("Duplicate: " + a);
      System.out.println(s.size()+" distinct words: "+s);
   }
}

java FindDups i came i saw i learned

Duplicate: i
Duplicate: i
4 distinct words: [i, learned, saw, came]
```

- Note that the code always refers to the Collection by its interface type (`Set`) rather than by its implementation type (`HashSet`).
- This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor.

Example: Modified Find Duplicate Strings

```
import java.util.*;
public class FindDups2 {
   public static void main(String args[]) {
       Set<String> uniques = new HashSet<String>();
      Set<String> dups = new HashSet<String>();
      for (String a : args)
            if (!uniques.add(a)) dups.add(a);

        //Destructive set-difference
        uniques.removeAll(dups);
        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
   }
}

java FindDups2 i came i saw i learned

Unique words:    [learned, saw, came]
Duplicate words: [i]
```

Concrete Collections Classes

- The concrete class `LinkedList<T>` is a concrete derived class of the abstract class `AbstractSequentialList<T>`
  - When efficient sequential movement through a list is needed, the `LinkedList<T>` class should be used
- The interface `SortedSet<T>` and the concrete class `TreeSet<T>` are designed for implementations of the `Set<T>` interface that provide for rapid retrieval of elements
  - The implementation of the class is similar to a binary tree, but with ways to do inserting that keep the tree balanced

The `LinkedList` Class

- It is a different means of achieving a scalar collection
  - Each element in a linked list is discrete in memory
  - Each element holds a reference to the next element and the previous one
- Linked lists behave well for insertions and deletions – no need to slide elements when adding
  - An existing link is broken and two new ones are formed
  - Deletion is the opposite process
- Iterating is less fast
  - Cannot randomly seek, must walk from element to element

Pitfall: Omitting the <T>

- Omitting **`<T>`** or corresponding class name from a reference to a collection class is an error for which the compiler may or may not issue an error message (depending on the details of the code), and even if it does, the error message may be quite strange
- Look for a missing **`<T>`** or **`<ClassName>`** when a program that uses collection classes gets a strange error message or doesn't run correctly

A Peek at the Map Framework

- The Java *map* framework deals with collections of *ordered pairs*
  - For example, a key and an associated value
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes
- The map framework uses the **`Map<T>`** interface, the **`AbstractMap<T>`** class, and classes derived from the **`AbstractMap<T>`** class

Enumerations & Iterators

- They are objects used to step through a container.
  - Available for some standard container classes which implement the corresponding interfaces.
  - Work properly even if the container changes.
  - Order might or might not be significant.
- Java has two variations:
  - **`Enumeration`** (old: since JDK 1.0)
  - **`Iterator`** (new: since JDK 1.2)

Enumerations

- To get an Enumeration `e` for container `v`:
  - `Enumeration e = v.elements();`
  - This is initialized at the start of the list.
- To get the first and subsequent elements:
  - `someObject = e.nextElement()`
- To test if all have been accessed:
  - `e.hasMoreElements()`
- Example:

```
for (Enumeration e = v.elements();
  e.hasMoreElements(); ) {
    System.out.println(
        e.nextElement());
}
```

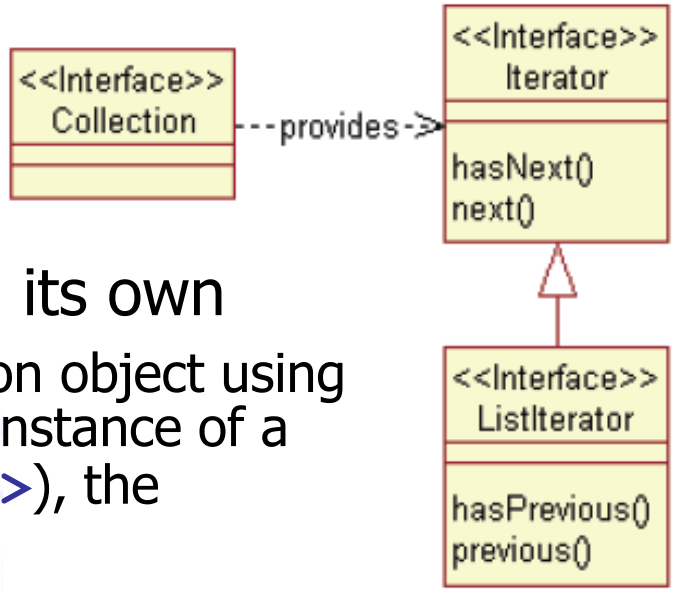
Iterators

- An *iterator* is an object that is used with a collection to provide sequential access to the collection elements
  - This access allows examination and possible modification of the elements
- An iterator imposes an ordering on the elements of a collection even if the collection itself does not impose any order on the elements it contains
  - If the collection does impose an ordering on its elements, then the iterator will use the same ordering

The `Iterator<T>` Interface

- The `Iterator<T>` interface isolates the use of a collection from the collection class itself

```
interface Iterator
{
    public boolean hasNext();
    public Object next();
    public void remove();// optional
}
```



- An `Iterator<T>` does not stand on its own
  - It must be associated with some collection object using the method `iterator`. E.g. If `c` is an instance of a collection class (e.g., `HashSet<String>`), the following obtains an iterator for `c`:

```
Iterator iteratorForC = c.iterator();
```

Using an Iterator with a HashSet<T> Object

- A `HashSet<T>` object imposes no order on the elements it contains
- However, an iterator will impose an order on the elements in the hash set
  - That is, the order in which they are produced by `next()`
  - Although the order of the elements so produced may be duplicated for each program run, there is no requirement that this must be the case

Example: An Iterator on `HashSet<T>`

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>( );
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        Iterator<String> i = s.iterator( );
        while (i.hasNext( )) System.out.println(i.next( ));
        i.remove( );
        System.out.println( );
        System.out.println("The set now contains:");
        i = s.iterator( );
        while (i.hasNext( )) System.out.println(i.next( ));
        System.out.println("End of program.");
    }
}
```

Tip: For-Each Loops as Iterators

- Although it is not an iterator, a for-each loop can serve the same purpose as an iterator
  - A for-each loop can be used to cycle through each element in a collection
- For-each loops can be used with any of the collections discussed here
- Ordinary `for` loops cannot cycle through the elements in a collection object
  - Unlike array elements, collection object elements are not normally associated with indices
- Although an ordinary `for` loop cannot cycle through the elements of a collection class, an enhanced `for` loop can cycle through the elements of an array

The "for each" Loop

- The general syntax for a **for**-each loop statement used with a collection

```
for (CollectionType VariableName :
  CollectionName)
  Statement
```

- The above **for**-each line should be read as "for each **VariableName** in **CollectionName**" do the following.
  - Note that **VariableName** must be declared within the **for**-each loop, not before
  - Note also that a colon (not a semicolon) is used after **VariableName**

Example: For-Each Loops as Iterators

```
import java.util.HashSet;
import java.util.Iterator;
public class ForEachDemo {
    public static void main(String[] args) {
        HashSet<String> s = new HashSet<String>( );
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        String last = null;
        for (String e : s) {
           last = e;
           System.out.println(e);
        }
        s.remove(last);
        System.out.println( );
        System.out.println("The set now contains:");
        for (String e : s) System.out.println(e);
        System.out.println("End of program.");
    }
}
```

Using Generics

- A *generic type* is defined in terms of some other type which it collects or on which it acts in some way, using angle brackets (<>)
  - Example: an **ArrayList<Point>** is an array-list of **Point** objects (from the **java.awt** package)

```
ArrayList<Point> someList = new ArrayList<Point>();
```

  - allows the compiler to catch a mistake like:

```
somelist.add(new Dimension(10, 10));
```

  - A type-specific collection object will provided a type specific iterator

```
Iterator<Point> each = someList.iterator();
```

  - Then it is no longer necessary to downcast the results of accessor methods, e.g.

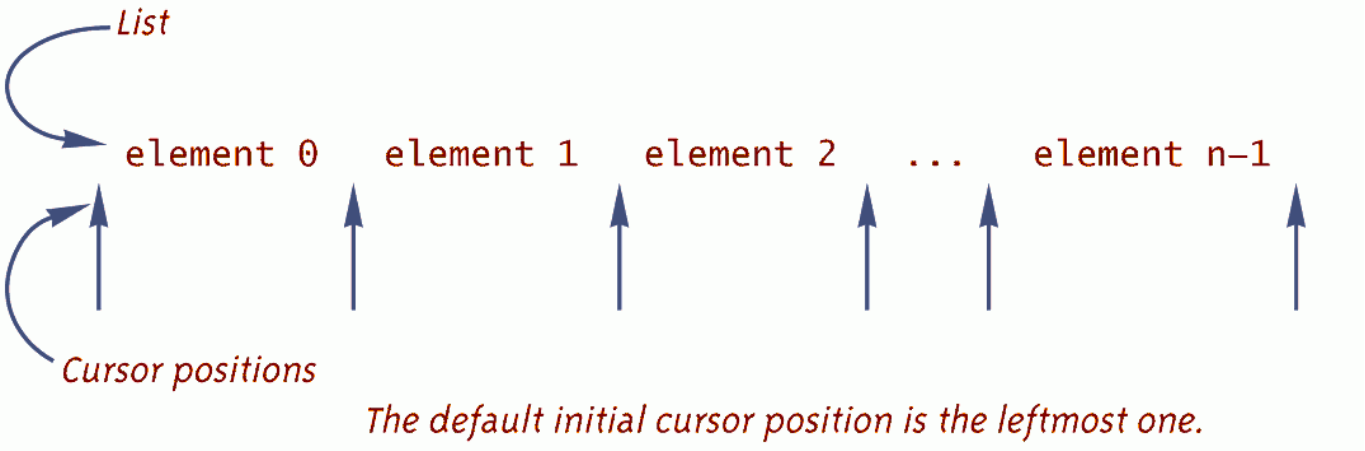
```
while (each.hasNext())
   each.next().x = 11;
```

The `ListIterator<T>` Interface

- The `ListIterator<T>` interface extends the `Iterator<T>` interface, and is designed to work with collections that satisfy the `List<T>` interface
  - A `ListIterator<T>` has all the methods that an `Iterator<T>` has, plus additional methods
  - A `ListIterator<T>` can move in either direction along a list of elements
  - A `ListIterator<T>` has methods, such as `set()` and `add()`, that can be used to modify elements

The `ListIterator<T>` Cursor

- Every `ListIterator<T>` has a position marker known as the *cursor*
  - If the list has *n* elements, they are numbered by indices 0 through *n*-1, but there are *n*+1 cursor positions
  - When `next()` is invoked, the element immediately following the cursor position is returned and the cursor is moved forward one cursor position
  - When `previous()` is invoked, the element immediately before the cursor position is returned and the cursor is moved back one cursor position



Pitfall: `next` and `previous` Can Return a Reference

- Theoretically, when an iterator operation returns an element of the collection, it might return a copy or clone of the element, or it might return a reference to the element
- Iterators for the standard predefined collection classes, such as `ArrayList<T>` and `HashSet<T>`, actually return references
  - Therefore, modifying the returned value will modify the element in the collection

Tip: Defining Your Own Iterator Classes

- There is usually little need for a programmer defined `Iterator<T>` or `ListIterator<T>` class
- The easiest and most common way to *define* a collection class is to *make it a derived class* of one of the library collection classes
  - By doing this, the `iterator()` and `listIterator()` methods automatically become available to the program
- If a collection class must be defined in some other way, then an iterator class should be defined as an inner class of the collection class

Reading

- Eckel: chapter 16, chapter 18
- Barnes: chapter 4
- Deitel: chapter 23, chapter 20

Summary

- Applet
  - differences from standalone applications
  - applet methods
  - parameters
  - packages and classes with applets
  - dual application/applet
  - limitations
  - Swing with applets
- Java Collections
  - arrays vs collections
  - API Collections
  - wildcards, generics
  - building collections
  - concrete collection classes:
    - ArrayList
    - Vector
    - LinkedList
    - HashSet
    - SortedSet
  - Iterators
  - The for-each loop