Abstract Data Type
Lecture 2

15 November 2011

Agenda

- Basic JAVA I/O
- JAVA IDEs, Notepad, Net beans, Eclipse
- JAVA Collection
- Pseudo code

Eclipse

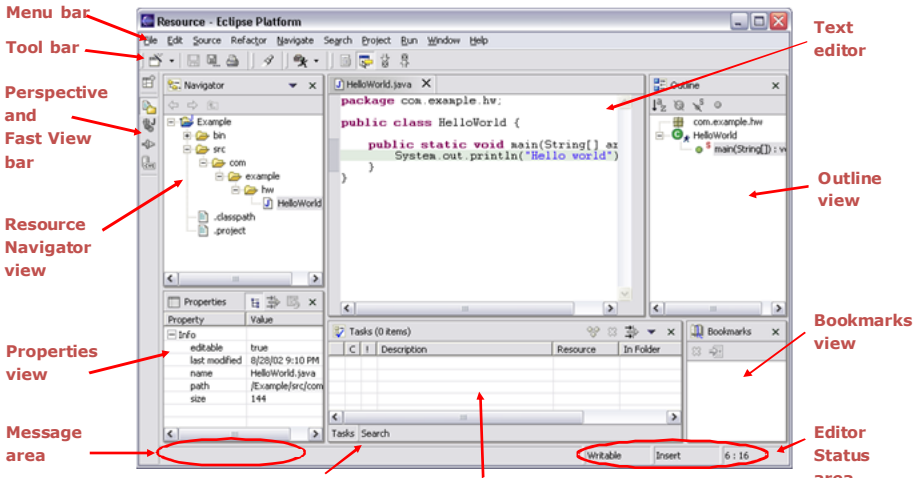
9-Dec-11

About IDEs

- An IDE is an Integrated Development Environment
- Different IDEs meet different needs
  - BlueJ, DrJava are designed as teaching tools
    - Emphasis is on ease of use for beginners
    - Little to learn, so students can concentrate on learning Java
  - Eclipse, JBuilder, NetBeans are designed as professional-level work tools
    - Emphasis is on supporting professional programmers
    - More to learn, but well worth it in the long run
- We will use Eclipse, but other professional IDEs are similar
- The following slides are taken from
  www.eclipse.org/eclipse/presentation/eclipse-slides.ppt

4

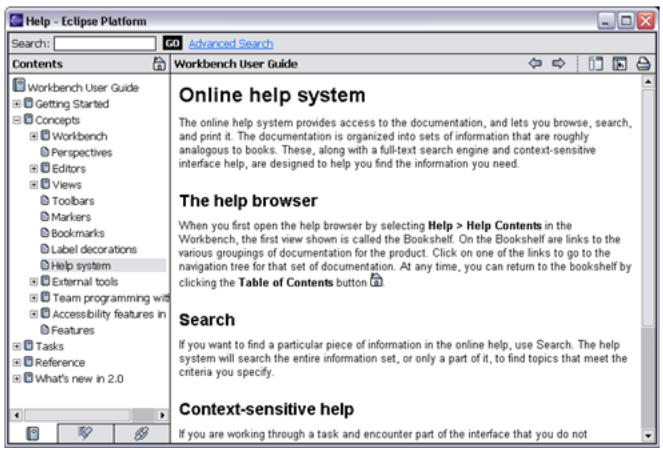
Workbench Terminology



5

Help Component

- Help is presented in a standard web browser



6

Java Development Tools

- JDT = Java development tools
- State of the art Java development environment

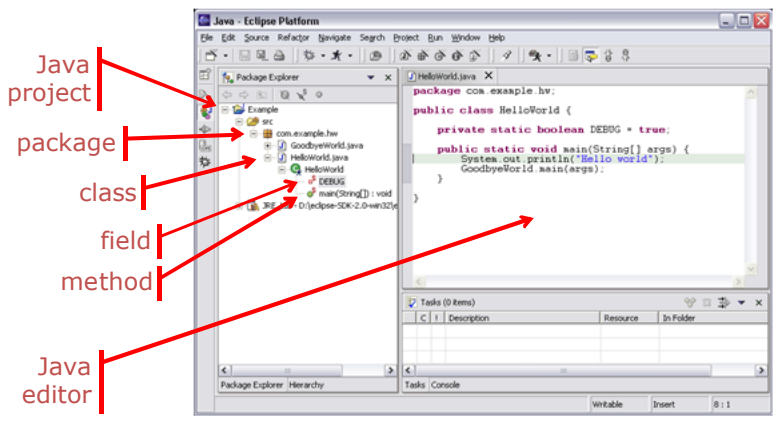
- Built atop Eclipse Platform
  - Implemented as Eclipse plug-ins
  - Using Eclipse Platform APIs and extension points

- Included in Eclipse Project releases
  - Available as separately installable feature
  - Part of Eclipse SDK drops

7

Java Perspective

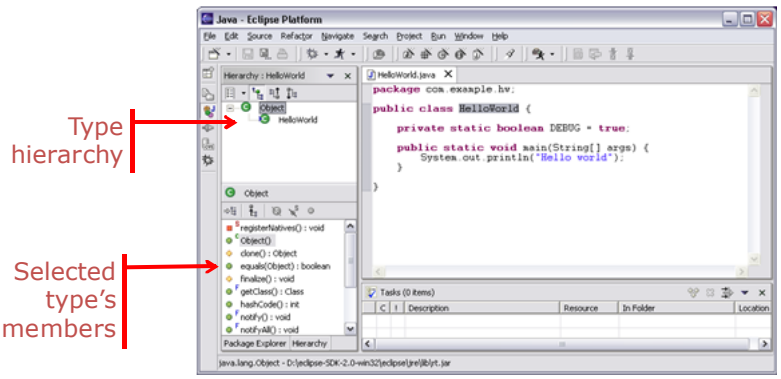
- Java-centric view of files in Java projects
  - Java elements meaningful for Java programmers



8

Java Perspective

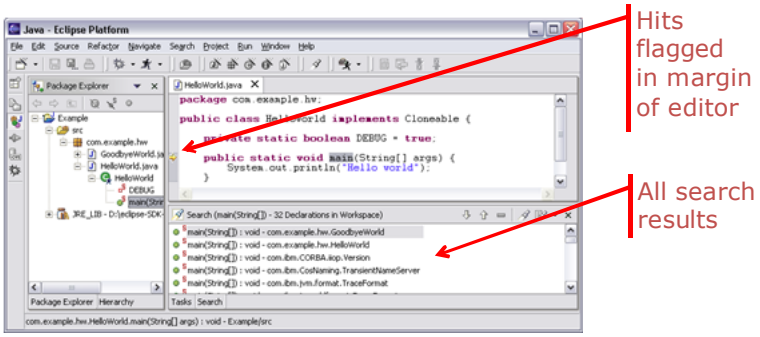
- Browse type hierarchies
  - "Up" hierarchy to supertypes
  - "Down" hierarchy to subtypes



9

Java Perspective

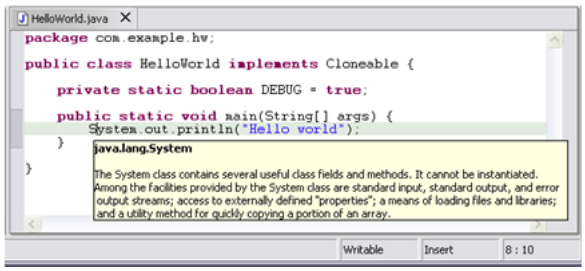
- Search for Java elements
  - Declarations or references
  - Including libraries and other projects



10

Java Editor

- Hovering over identifier shows Javadoc spec



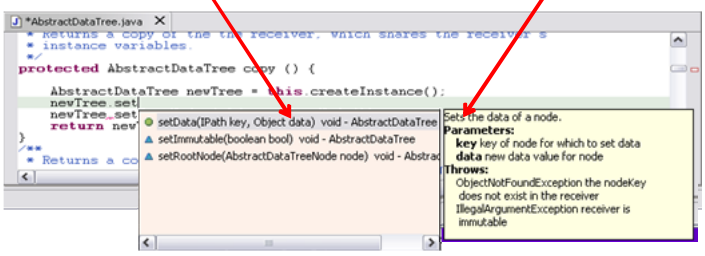
11

Java Editor

- Method completion in Java editor

List of plausible methods

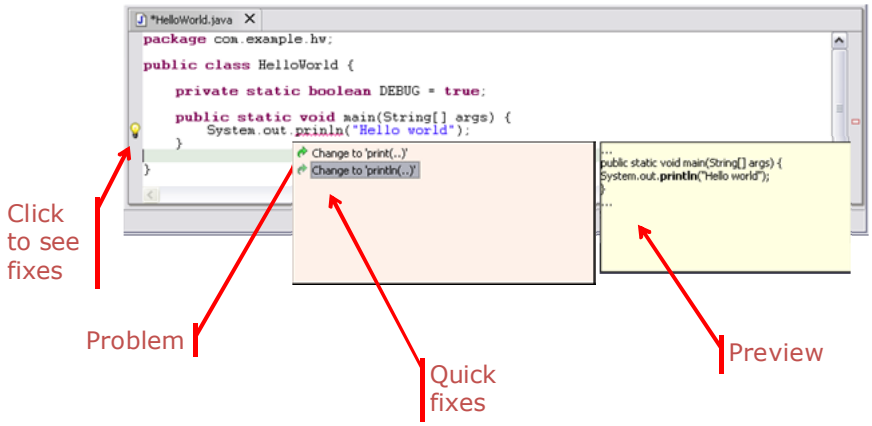
Doc for method



12

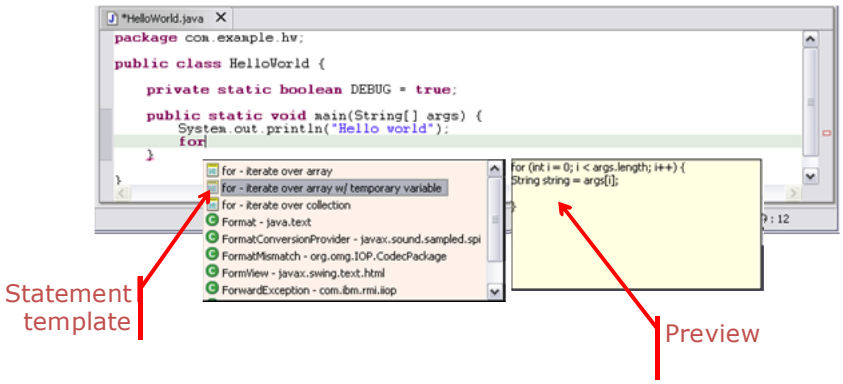
Java Editor

- On-the-fly spell check catches errors early



Java Editor

- Code templates help with drudgery

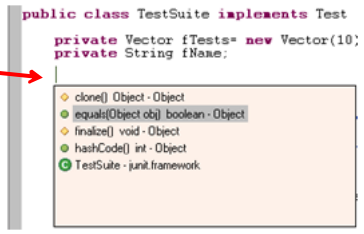


Java Editor

- Java editor creates stub methods

Method stub insertion for anonymous inner types

Method stub insertion for inherited methods

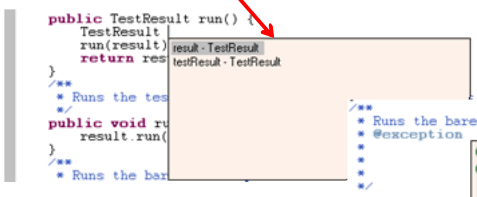


15

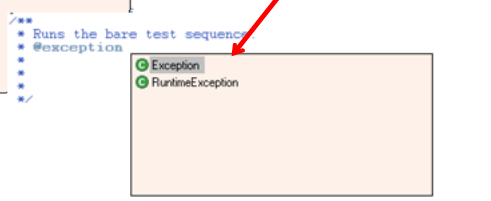
Java Editor

- Java editor helps programmers write good Java code

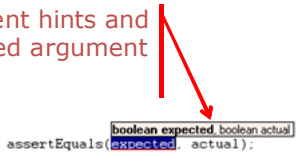
Variable name suggestion



JavaDoc code assist



Argument hints and proposed argument names



16

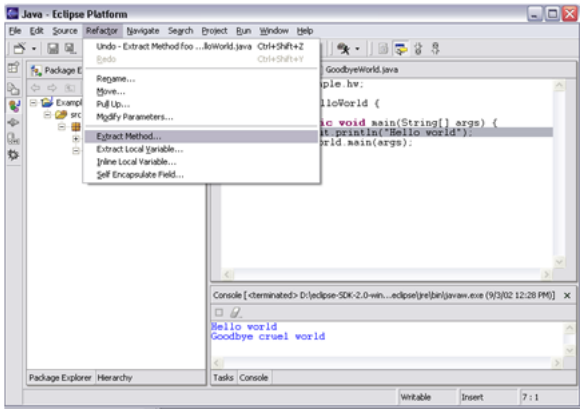
Java Editor

- Other features of Java editor include
  - Local method history
  - Code formatter
  - Source code for binary libraries
  - Built-in refactoring

17

Refactoring

- JDT has actions for refactoring Java code



18

Refactoring

- Refactoring actions rewrite source code
  - Within a single Java source file
  - Across multiple interrelated Java source files
- Refactoring actions preserve program semantics
  - Does not alter what program does
  - Just affects the way it does it
- Encourages exploratory programming
- Encourages higher code quality
  - Makes it easier to rewrite poor code

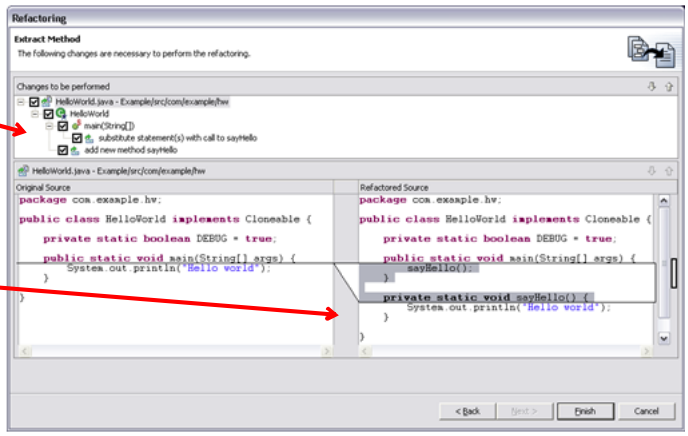
19

Refactoring

- Full preview of all ensuing code changes
  - Programmer can veto individual changes

List of changes

"before" vs. "after"



20

Refactoring

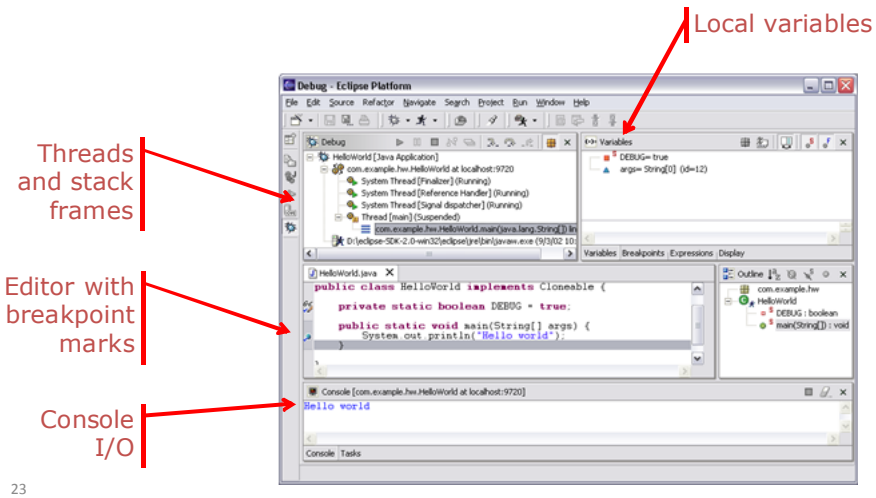
- Growing catalog of refactoring actions
  - Organize imports
  - Rename {field, method, class, package}
  - Move {field, method, class}
  - Extract method
  - Extract local variable
  - Inline local variable
  - Reorder method parameters

Eclipse Java Compiler

- Eclipse Java compiler
  - JCK-compliant Java compiler (selectable 1.3 and 1.4)
  - Helpful error messages
  - Generates runnable code even in presence of errors
  - Fully-automatic incremental recompilation
  - High performance
  - Scales to large projects
- Multiple other uses besides the obvious
  - Syntax and spell checking
  - Analyze structure inside Java source file
  - Name resolution
  - Content assist
  - Refactoring
  - Searches

Eclipse Java Debugger

- Run or debug Java programs



23

Eclipse Java Debugger

- Run Java programs
  - In separate target JVM (user selectable)
  - Console provides stdout, stdin, stderr
  - Scrapbook pages for executing Java code snippets
- Debug Java programs
  - Full source code debugging
  - Any JPDA-compliant JVM
- Debugger features include
  - Method and exception breakpoints
  - Conditional breakpoints
  - Watchpoints
  - Step over, into, return; run to line
  - Inspect and modify fields and local variables
  - Evaluate snippets in context of method
  - Hot swap (if target JVM supports)

24

The End

25

Pseudocode

An Introduction

Flowcharts were the first design tool to be widely used, but unfortunately they do not reflect some of the concepts of structured programming very well. Pseudocode, on the other hand, is a newer tool and has features that make it more reflective of the structured concepts. The drawback is that the narrative presentation is not as easy to understand and/or follow.

Rules for Pseudocode

- Write only one statement per line
  - Capitalize initial keyword
  - Indent to show hierarchy
  - End multiline structures
  - Keep statements language independent

One Statement Per Line

Each statement in pseudocode should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.

Task List	Pseudocode
Read name, hours worked, rate of pay	READ name, hoursWorked, payRate
Perform calculations	gross = hoursWorked * payRate
gross = hours worked * rate of pay	WRITE name, hoursWorked, gross
Write name, hours worked, gross	

Capitalize Initial Keyword

In the example below note the words: READ and WRITE. These are just a few of the keywords to use, others include:

READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE

Pseudocode

```
READ name, hoursWorked, payRate
gross = hoursWorked * payRate
WRITE name, hoursWorked, gross
```

Indent to Show Hierarchy

Each design structure uses a particular indentation pattern

- Sequence:
  Keep statements in sequence all starting in the same column
- Selection:
  Indent statements that fall inside selection structure, but not the keywords that form the selection
- Loop:
  Indent statements that fall inside the loop but not keywords that form the loop

```
READ name, grossPay, taxes
IF taxes > 0
        net = grossPay – taxes
ELSE
        net = grossPay
ENDIF
WRITE name, net
```

End Multiline Structures

```
READ name, grossPay, taxes
IF taxes > 0
      net = grossPay – taxes
ELSE
      net = grossPay
ENDIF
WRITE name, net
```

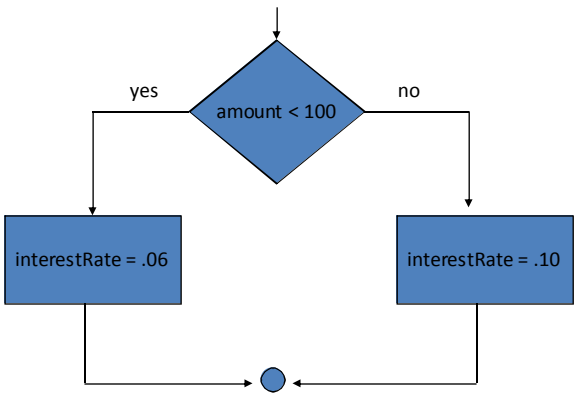
See the IF/ELSE/ENDIF as constructed above, the ENDIF is in line with the IF.

The same applies for WHILE/ENDWHILE etc…

Language Independence

Resist the urge to write in whatever language you are most comfortable with, in the long run you will save time. Remember you are describing a logic plan to develop a program, you are not programming!

The Selection Structure



Pseudocode ➔

```
IF amount < 100
        interestRate = .06
ELSE
        Interest Rate = .10
ENDIF
```

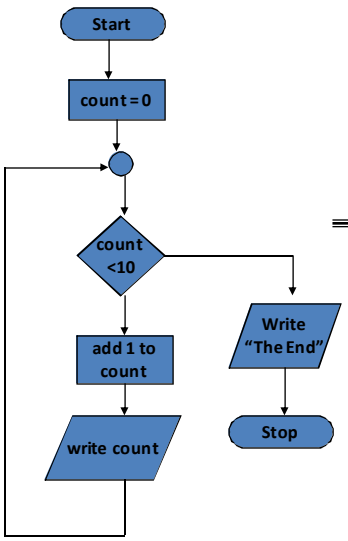
The Looping Structure

In flowcharting one of the more confusing things is to separate selection from looping. This is because each structure use the diamond as their control symbol. In pseudocode we avoid this by using specific keywords to designate looping

WHILE/ENDWHILE

REPEAT/UNTIL

WHILE / ENDWHILE




```
count = 0
WHILE count < 10
      ADD 1 to count
      WRITE count
ENDWHILE
WRITE "The End"
```

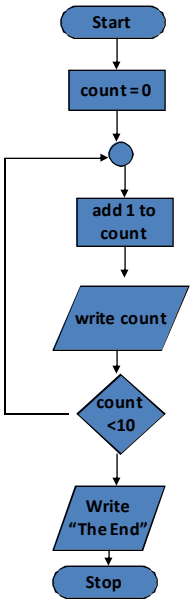
Mainline ← Modular

```
count = 0
WHILE count < 10
      DO Process
ENDWHILE
WRITE "The End"
```

Process

```
ADD 1 to count
WRITE count
```

REPEAT / UNTIL




```
count = 0
REPEAT
      ADD 1 to count
      WRITE count
UNTIL count >= 10
WRITE "The End"
```

Mainline ← Modular

```
count = 0
REPEAT
      DO Process
UNTIL count >= 10
WRITE "The End"
```

Process

```
ADD 1 to count
WRITE count
```

Advantages & Disadvantages

Flowchart Advantages:

- ✓ Standardized
- ✓ Visual

Pseudocode Advantages

- ✓ Easily modified
- ✓ Implements structured concepts
- ✓ Done easily on Word Processor

Flowchart Disadvantages:

- ✓ Hard to modify
- ✓ Structured design elements not implemented
- ✓ Special software required

Pseudocode Disadvantages:

- ✓ Not visual
- ✓ No accepted standard, varies from company to company

Access of Data

The READ statement tells the computer to get a value from an input device and store it in a memory location.

How to deal with memory locations?

Memory locations are identified by their addresses, we give them names (field names / variable names) using words descriptive to us such as ctr as opposed to a location addresses such as 19087.

Rules for Variable Names

- Begin with lowercase letter
- Contain no spaces
- Additional words begin with capital
- Unique names within code
- Consistent use of names

Working with Fields

Calculations		Selection	
+	add	>	greater than
-	subtract	<	less than
*	multiply	=	equal to
/	divide	>=	greater than or equal to
** or ^	exponentiation	<=	less than or equal to
( )	grouping	<>	not equal to

What is the Collections framework?

- Collections framework provides two things:
  - implementations of common high-level *data structures*: e.g. Maps, Sets, Lists, etc.
  - An organized class hierarchy with rules/formality for adding new implementations
- The latter point is the sense in which Collections are a *framework*.
- Note the difference between providing a framework + implementation and just implementation.
- Some other differences:
  - code reuse
  - clarity
  - unit testing?

Definition of collection

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- They typically represent data items that form a natural group, e.g.
  - poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).

History

- Pre Java SDK1.2, Java provided a handful of data structures:
  - Hashtable
  - Vector
  - Bitset
- These were for the most part good and easy to use, but they were not organized into a more general framework.
- SDK1.2 added the larger skeleton which organizes a much more general set of data structures.
- Legacy datastructures retrofitted to new model.
- *Generic types*/autoboxing added in 1.5

General comments about data structures

- "Containers" for storing data.
- Different data structures provide different abstractions for getting/setting elements of data.
  - linked lists
  - hashtables
  - vectors
  - arrays
- Same data structures can even be implemented in different ways for performance/memory:
  - queue over linked list
  - queue over arrays

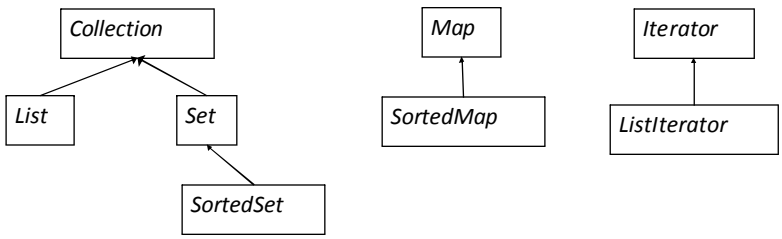
More on data structures

- Everyone should take a basic class in building data structures
- I recommend the book Mastering Algorthims with C by Kyle Loudon
- In Java, one does not usually build data structures, but rather uses the provided one
- Using Java's data structures requires a little understanding of the Collections framework
- Adding your own requires a deeper understanding.

Learning to use data structures

- Dual purposes for us to study Collections:
  - Be able to choose, properly use built-in data structures.
  - Another study in OO class design
- Thus, we start by study the Collections class design.
- Then, we provide many examples of how to use the built-in types in real programming.

Collections-related Interface hierarchy



- The *Collection* inteface stores groups of Objects, with duplicates allowed
- The *Set* interface extends *Collection* but forbids duplicates
- The *List* interface extends *Collection*, allows duplicates, and introduces positional indexing.
- *Map* is a separate hierarchy

Collection implementations

- Note that Java does not provide any direct implementations of *Collection*.
- Rather, concrete implementations are based on other interfaces which extend Collection, such as Set, List, etc.
- Still, the most general code will be written using Collection to type variables.

A Peek at generics

Old way

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

New way with Generics …

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Generics vs. Casting

- Note that new method is backward compatible with old method
  - Old code ports
  - Warnings issues by compiler
- What are some advantages of Generics?
- What are some disadvantages?
- How can we use Generics if we are mixing types?

Another example of Generics

Here is a simple example taken from the existing Collections tutorial:

```
// Removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext(); )
    if (((String) i.next()).length() == 4) i.remove();
}
```

Here is the same example modified to use generics:

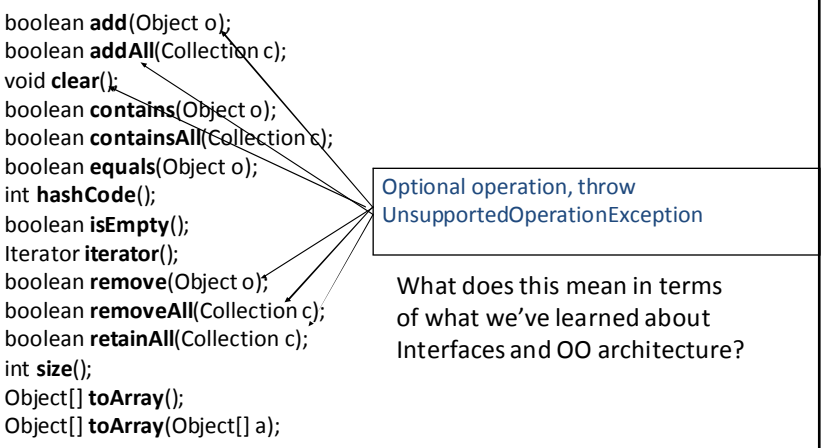
Think "Collection of Strings"

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
  for (Iterator<String> i = c.iterator(); i.hasNext(); )
    if (i.next().length() == 4) i.remove();
}
```

Generics

- There are lots of little subtleties introduced by this capability.
- Better not to dwell on them this week – will pick up after we study the Collections themselves.

Collection Interface



What does this mean in terms of what we've learned about Interfaces and OO architecture?

Comments on Collection methods

- Note the iterator() method, which returns an Object which implements the *Iterator* interface.
- *Iterator* objects are used to traverse elements of the collection in their natural order.
- Iterator has the following methods:
  - boolean hasNext(); // are there any more elements?
  - Object next(); // return the next element
  - void remove(); // remove the element returned after lest next()

AbstractCollection Class

java.util.*AbstractCollection*

- Abstract class which is partial implementation of of Collection interface
- Implements all methods except iterator() and size()
- Makes it much less work to implement Collections Interface

List interface

- An interface that extends the Collections interface.
- An ordered collection (also known as a *sequence*).
  - The user of this interface has precise control over where in the list each element is inserted.
  - The user can access elements by their integer index (position in the list), and search for elements in the list.
- Unlike *Set*, allows duplicate elements.
- Provides a special *Iterator* called *ListIterator* for looping through elements of the List.

Additional methods in *List* Interface

- *List* extends *Collection* with additional methods for performing index-based operations:
  - void add(int index, Object element)
  - boolean addAll(int index, Collection collection)
  - Object get(int index)
  - int indexOf(Object element)
  - int lastIndexOf(Object element)
  - Object remove(int index)
  - Object set(int index, Object element)

List/ListIterator Interface

- The List interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position friendly manner:
  - ListIterator listIterator()
  - ListIterator listIterator(int startIndex)
  - List subList(int fromIndex, int toIndex)
- *ListIterator* extends *Iterator* and adds methods for bi-directional traversal as well as adding/removing elements from the underlying collection.

Randomly shuffling a List

```
import java.util.*;

public class Shuffle {
   public static void main(String[] args) {
      List<String> list = Arrays.asList(args);
      Collections.shuffle(list);
      System.out.println(list);
   }
}
```

Concrete List Implementations

- There are two concrete implementations of the *List* interface
  - LinkedList
  - ArrayList
- Which is best to use depends on specific needs.
- Linked lists tend to be optimal for inserting/removing elements.
- ArrayLists are good for traversing elements sequentilly
- Note that LinkedList and ArrayList both extend abstract partial implementations of the *List* interface.

LinkedList Class

- The LinkedList class offeres a few additional methods for directly manipulating the ends of the list:
  - void addFirst(Object)
  - void addLast(Object);
  - Object getFirst();
  - Object getLast();
  - Object removeFirst();
  - Object removeLast();
- These methods make it natural to implement other simpler data structures, like Stacks and Queues.

LinkedList examples

- See heavily commented LinkedList Example in course notes
- A few things to be aware of:
  - it is really bad to use the positional indexing features copiously of LinkedList if you care at all about performance. This is because the LinkedList has no memory and must always traverse the chain from the beginning.
  - Elements can be changed both with the List and ListIterator objects. That latter is often more convenient.
  - You can create havoc by creating several iterators that you use to mutate the List. There is some protection built-in, but best is to have only one iterator that will actually mutate the list structure.

ArrayList Class

- Also supports the List interface, so top-level code can pretty much invisibly use this class or LinkedList (minus a few additional operations in LinkedList).
- However, ArrayList is much better for using positional index access methods.
- At the same time, ArrayList is much worse at inserting elements.
- This behavior follows from how ArrayLists are structured: they are just like Vectors.

More on ArrayList

- Additional methods for managing size of underlying array
- *size*, *isEmpty*, *get*, *set*, *iterator*, and *listIterator* methods all run in constant time.
- Adding n elements take O[n] time.
- Can explicitly grow capacity in anticipation of adding many elements.
- Note: legacy Vector class almost identical. Main differences are naming and synchronization.
- See short **ArrayList** example.

Vector class

- Like an ArrayList, but synchronized for multithreaded programming.
- Mainly for backwards-compatibility with old java.
- Used also as base class for *Stack* implementation.

Stack class

- **Stack**()
  Creates an empty Stack. **Method**
- boolean **empty**()
  Tests if this stack is empty.
- E **peek**()
  Looks at the object at the top of this stack without removing it from the stack.
- E **pop**()
  Removes the object at the top of this stack and returns that object as the value of this function.
- E **push**(E item)
  Pushes an item onto the top of this stack.
- int **search**(Object o)
  Returns the 1-based position where an object is on this stack.

```
public class Deal {
  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("Usage: Deal hands cards");
      return;
    }
    int numHands = Integer.parseInt(args[0]);
    int cardsPerHand = Integer.parseInt(args[1]);

    // Make a normal 52-card deck.
    String[] suit = new String[] {
       "spades", "hearts", "diamonds", "clubs" };
    String[] rank = new String[] {
       "ace","2","3","4","5","6","7","8",
       "9","10","jack","queen","king" };
    List<String> deck = new ArrayList<String>();
    for (int i = 0; i < suit.length; i++)
      for (int j = 0; j < rank.length; j++)
        deck.add(rank[j] + " of " + suit[i]);

    // Shuffle the deck.
    Collections.shuffle(deck);

    if (numHands * cardsPerHand > deck.size()) {
      System.out.println("Not enough cards.");
      return;}

    for (int i=0; i < numHands; i++)
      System.out.println(dealHand(deck, cardsPerHand));}

  public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;}
}
```

Set Interface

- Set also extends *Collection*, but it prohibits duplicate items (this is what defines a Set).
- No new methods are introduced; specifically, none for index-based operations (elements of Sets are not ordered).
- Concrete Set implementations contain methods that forbid adding two equal Objects.
- More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element
- Java has two implementations: HashSet, TreeSet

Using Sets to find duplicate elements

```
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 detected: " + a);

      System.out.println(s.size() + " distinct words: " + s);
   }
}
```

HashSets and hash tables

- Lists allow for ordered elements, but searching them is very slow.
- Can speed up search tremendously if you don't care about ordering.
- Hash tables let you do this. Drawback is that you have no control over how elements are ordered.
- hashCode() computes integer (quickly) which corresponds to position in hash table.
- Independent of other objects in table.

HashSet Class

- Hashing can be used to implement several important data structures.
- Simplest of these is HashSet
  - add elements with add(Object) method
  - contains(Object) is redefined to first look for duplicates.
  - if duplicate exists, Object is not added
- What determines a duplicate?
  - careful here, must redefine both hashCode() and equals(Object)!

HashSet

- Look HashSetExample.java
- Play around with some additional methods.
- Try creating your own classes and override hashCode method.
- Do Some timings.

Tree Sets

- Another concrete set implementation in Java is TreeSet.
- Similar to HashSet, but one advantage:
  - While elements are added with no regard for order, they are returned (via iterator) in sorted order.
  - What is sorted order?
    - this is defined either by having class implement *Comparable* interface, or passing a *Comparator* object to the TreeSet Constructor.
    - Latter is more flexible: doesn't lock in specific sorting rule, for example. Collection could be sorted in one place by name, another by age, etc.

Comparable interface

- Many java classes already implement this. Try String, Character, Integer, etc.
- Your own classes will have to do this explicitly:
  - *Comparable* defines the method
    public int compareTo(Object other);
  - *Comparator* defines the method
    public int compare(Object a, Object b);
- As we discussed before, be aware of the general contracts of these interfaces.
- See TreeSetExample.java

Maps

- Maps are similar to collections but are actually represented by an entirely different class hierarchy.
- Maps store objects by key/value pairs:
  - map.add("1234", "Andrew");
  - ie Object Andrew is stored by Object key 1234
- Keys may not be duplicated
- Each key may map to only one value

Java *Map* interface

- Methods can be broken down into three groups:
  - querying
  - altering
  - obtaining different views
- Fairly similar to Collection methods, but Java designers still thought best to make separate hierarchy – no simple answers here.

Map methods

- Here is a list of the Map methods:
  - void clear()
  - boolean containsKey(Object)
  - boolean containsValue(Object)
  - Set entrySet()
  - boolean get(Object)
  - boolean isEmpty()
  - Set keySet()
  - Object put(Object, Object)
  - void putall(Map)
  - Object remove(Object)
  - int size()
  - Collection values()

Map Implementations

- We won't go into too much detail on Maps.
- Java provides several common class implementations:
  - HashMap
    - a hashtable implementation of a map
    - good for quick searching where order doesn't matter
    - must override hashCode and equals
  - TreeMap
    - A tree implementation of a map
    - Good when natural ordering is required
    - Must be able to define ordering for added elements.

Vector

- What is Java **Vector** Class?
- Class **Vector** constructors
- Class **Vector** methods
- Interface **Enumeration** and its methods

What is Java **Vector** Class?

- **Java class Vector provides the capabilities of array-like data structures that can dynamically resize themselves.**
- **At any time the Vector contains a certain number of elements which can be different types of objects and the size is less than or equal to its *capacity*.**
- **The *capacity* is the space that has been reserved for the array.**
- **If a Vector needs to grow, it grows by an increment that you specify.**
- **If you do not specify a capacity increment, the system will automatically double the size of the Vector each time additional capacity is needed.**

Class **Vector** constructors

public Vector();

Constructs an empty vector so that its internal data array has size 10.

e.g. Vector v = new Vector();

public Vector(int initialCapacity);

Constructs an empty vector with the specified initial capacity.

e.g. Vector v = new Vector(1);

public Vector(int initialCapacity, int capacityIncrement);

Constructs an empty vector with the specified initial capacity and capacity increment.

e.g. Vector v = new Vector(4, 2);

Class **Vector** methods

public void addElement(Object obj)

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

e.g. v.addElement(input.getText());

public boolean removeElement(Object obj)

Removes the first (lowest-indexed) occurrence of the argument from this vector. If the object is found in this vector, each component in the vector with an index greater or equal to the object's index is shifted downward to have an index one smaller than the value it had previously.

e.g. if (v.removeElement(input.getText()))

showStatus("Removed: " + input.getText());

Class **Vector** methods

public Object firstElement()

Returns the first component (the item at index 0) of this vector.

e.g. showStatus(v.firstElement());

public Object lastElement()

Returns the last component of the vector.

e.g. showStatus(v.lastElement());

public boolean isEmpty()

Tests if this vector has no components.

e.g. showStatus(v.isEmptyt()? "Vector is empty" : "Vector is not empty");

Class **Vector** methods

public boolean contains(Object elem)

Tests if the specified object is a component in this vector.

e.g. if (v.contains(input.getText()))

showStatus("Vector contains: " + input.getText());

public int indexOf(Object elem)

Searches for the first occurence of the given argument, testing for equality using the equals method.

e.g. showStatus("Element is at location" + v.indexOf(input.getText()));

public int size()

Returns the number of components in this vector.

e.g. showStatus("Size is " + v.size());

Class **Vector** methods

public int capacity()

Returns the current capacity of this vector.

e.g. showStatus("Capacity is " + v.capacity());

public void trimToSize()

Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field elementData, with a smaller one. An application can use this operation to minimize the storage of a vector.

e.g. v.trimToSize();

public Enumeration elements()

Enumeration
列挙,一覧表

Returns an enumeration of the components of this vector. The returned Enumeration object will enumerate all items in this vector. The first item generated is the item at index 0, then the item at index 1, and so on.

Interface **Enumeration** and its methods

public abstract interface Enumeration

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series.

e.g. Enumeration enum = v.elements();

public Object nextElement()

Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

public boolean hasMoreElements()

Tests if this enumeration contains more elements.

e.g. While (enum.hasMoreElements())

showStatus(enum.nextElement());

An example of using class Vector

Run Java applet => VectorTest.html

Take a look at source code => VectorTest.java

Exercise:

1. Understand and run the program.

2. Add a reverse button and it displays the elements in the vector in a reverse order.

Optional exercise:

Write a java application program that uses Vector class and can output the result as shown in the right.

Output:

```
Enter lines of input, use quit to end the program.
apple
orange
banana
grape
lemon
quit
Number of lines: 5
Middle line:  banana
Lines in reverse order:
lemon
grape
banana
orange
apple
Lines in reverse alphabetical order:
orange
lemon
grape
banana
apple
```