Containers

- Hold a group of objects
- Significantly increase your programming power
- All perform bound checking
- array: efficient, can hold primitives
- Collection: a group of individual elements
  - List, Set
- Map: a group of key-value object pairs
  - HashMap
- Misleading: sometimes the whole container libraries are also called collection classes
array

- Most efficient way to do random access
- Size is fixed and cannot be changed for the lifetime
- If run out of space, have to create a new one and copy everything
- Advantage: can hold primitives
Other containers

- Can only take object
- Have to “wrap” primitives
  - int -> Integer, double-> Double
- Have to cast or unwrap on retrieval
- Slow, error prone, tedious….
- Fixed by JDK1.5, hopefully
- Advantage: automatic expanding
Arrays class

- In java.util, a “wrapper” class for array
- A set of static utility methods
  - `fill()`: fill an array with a value
  - `equals()`: compare two arrays for equality
  - `sort()`: sort an array
  - `binarySearch()`: find one element in a sorted array
- All these methods overload for all primitive types and Object
Arrays.sort()

- Sorts the objects into ascending order, according to their *natural ordering*
- This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort
- You can specify a range. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive.
- The objects need to *comparable* or there is a special *comparator*
Arrays.sort() cont.

- sort(array), sort(array, fromIndex, toIndex)
- All elements in the array must implement the `Comparable` interface

- sort(array, comparator)
- sort(array, fromIndex, toIndex, comparator)
- All elements in the array must be mutually comparable by the specified comparator
Comparable interface

- With a single method `compareTo()`
- Takes another Object as argument
- And returns:
  - Negative value if `this` is less than argument
  - Zero value if `this` is equal to argument
  - Positive value if `this` is greater than argument
Comparator interface

- Two methods: `compare()`, `equals()`
- Only need to implement `compare()`
- Takes two `Object` as argument: `compare(Object o1, Object o2)`
- And returns
  - Negative value if `o1` is less than `o2`
  - Zero value if `o1` is equal to `o2`
  - Positive value if `o1` is greater than `o2`
Sort example: WorkerTest.java
Array.binarySearch()

- Only usable on sorted array!
  - Otherwise, result unpredictable
- If there are multiple elements equal to the specified object, there is no guarantee which one will be found.
- Return:
  - Location if find the key (positive number)
  - \((-{(insertion\ point)} - 1)\) if not find key (negative)
search example: compareExp.java
Collection: hold one item at each location
List: items in order
Set: no duplicates, no ordering

Preserve the insertion of the elements
Map: key-value pairs, fast retrieval
no duplicate keys, no ordering
Disadvantages of container

- Cannot hold primitives
  - Have to wrap it
- Lose type information when put object into container
  - Everything is just Object type once in container
- Have to do cast when get it out
  - You need to remember what’s inside
- Java do run time type check
  - ClassCastException
ArrayList

- An array that automatically expand itself
- Put objects using add()
- Get out using get(int index)
  - Need to cast type
- Method size() to get the number of objects
  - Similar to .length attribute of array
- Example: CatsAndDogs.java
Iterator object

- Access method regardless of the underlying structure
- Generic programming
  - Can change underlying structure easily
- “light-weight” object
  - Cheap to create
- Can move in only one direction
Iterator constraints

- Container.iterator() returns you an Iterator, which is ready to return the first element in the sequence on your first call to next()
- Get the next object in the sequence with next()
- Set there are more objects in the sequence with hasNext()
- Remove the last element returned by the iterator with remove()
- Example: revisit CatsAndDogs.java
ArrayList vs. LinkedList

- **ArrayList**
  - Rapid random access
  - Slow when inserting or removing in the middle

- **LinkedList**
  - Optimal sequential access
  - Fast insertion and deletion from the middle
  - `addFirst()`, `addLast()`, `getFirst()`, `removeFirst()`
  - Easy to be used as queue, stack
Set interface

- Each element added to the Set must be unique, otherwise won’t add.
- Objects added to Set must define equals() to establish object uniqueness
- Not maintain order
HashSet
- Fast lookup time by hashing function

TreeSet
- Ordered Set backed by a tree (red-black tree)
- Can extract ordered sequence

LinkedHashSet
- Has the lookup speed of a HashSet
- Maintain the insertion order by linked list
Set example

- revisit CatsAndDogs.java
Map interface

- Key-value associative array
- Look up object using another object
  - Array use index
- put(Object key, Object value)
- Object get(Object key)
- containsKey(), containsValue()
Map

- HashMap
  - Based on a hash table
  - Constant time for insertion and locating pairs
  - Most commonly used

- LinkedHashMap
  - Like HashMap
  - When iterate through, get pairs in insertion order

- TreeMap
  - Based on red-black tree, can viewed in order
  - Need to implement Comparable or Comparator
Map example

MapExample.java
I/O

- Difficult task
- Too many sources and sinks to cover
  - File
  - Console
  - Network
- Many format
  - Binary-oriented
  - Unicode
  - Zipped
Java I/O: lowest level abstraction

- InputStream/OutputStream class
  - Byte-oriented
  - read() return the byte got read
  - write(int b) writes one byte out
- Can obtain from console, file, or socket
- Handle them in essentially the same way
- Write your code to work with Streams and won’t care if it’s talking to a file or a system on the other side of the world
Unicode

- Internationalization-friendly text representation
- Reader = InputStream for Unicode
- Write = OutputStream for Unicode
- Can build Reader from InputStream or sometimes get directly from source
  - Same for Writer
- Reader r = InputStreamReader(istream);
Stream from file

- Create from String
  ```java
  FileInputStream fin = new FileInputStream("data.txt")
  ```

- Or create a File object first
  ```java
  File f = new File("data.txt");
  FileInputStream fin = new FileInputStream(f);
  ```

- Note: can also create Files from directory
  ```java
  File f = new File("/home/kewang");
  File[] dirListing = f.listFiles();
  ```
Higher-level features

- Buffered
  - Higher performance through buffers
  - Allows readline() for Readers

- Pushback
  - Can “unread” a byte or character
  - InputStream, Reader
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.println("What’s your name?");
String name = "";
try{
    name = console.readLine();
}catch(IOException e) {} 
System.out.println("My name is: "+name);
Try:

```java
File f = new File("myfile.jpg");
int filesize = (int)f.length(); // get the file size
byte[] data = new byte[filesize];
//a stream to read the file
DataInputStream in = new DataInputStream(new FileInputStream(f));
in.readFully(data); //read file contents in array
in.close(); //remember to close it
```
catch( IOException e ) {}
try{
    BufferedReader in = new BufferedReader(new FileReader(filename));
    String line="";
    while( (line=in.readLine())!=null ) { //read the file line by line
        .... // process the read in line
    }
    in.close();
}
catch(IOException e){}
try{
    File f = new File(filename);
    PrintWriter out = new PrintWriter(new FileWriter(f));
    out.println("my filename is: "+filename);
    out.close();
}catch(IOException e){}

System.out just refers to the standard output stream
println(), print() can be used with any Writer.
StringTokenizer class

- Extract tokens from a string
- `StringTokenizer(String s, String delim)`
  - `s`: the input string from which tokens are read
  - `delim`: the delimiter character (any one in it is a delimiter)
  - Default delimiter is " 	
- boolean `hasMoreTokens()`
  - Return true if more token exists
- `String nextToken()`
  - Return the next token
String line = is.readLine();

//suppose readback a line: Mike|15.5|40
StringTokenizer st = new StringTokenizer(line, "|");

String name = st.nextToken();
double rate = Double.parseDouble(st.nextToken());
int hours = Integer.parseInt(st.nextToken());
Serialization

- An important feature of Java
- Convert an object into a stream of byte, and can later deserialize it into a copy of the original object
- Takes care of reassembling objects
- Need to cast type when read back
- Any object as long as it implements `Serializable` interface
  - No method inside
Employee e = new Employee();
try{
    //Serialize the object
    ObjectOutputStream oos = new ObjectOutputStream( new FileoutputStream(filename));
    oos.writeObject(e);
    oos.close();

    //read back the object
    ObjectInputStream ois = new ObjectInputStream( new FileInputStream(filename));
    Employee e2 = (Employee)ois.readObject();
    ois.close();
}
catch(IOException e) {}  
catch(ClassNotFoundException e) {} //readObject can throw this

//the class must implement the Serializable interface
class Employee implements Serializable{ … …}
Java networking

- Java.net.
- Create a socket with the internet address and port of the destination
  - Socket s = new Socket("www.columbia.edu", 80);
- Grab the associated streams
  - InputStream is = s.getInputStream();
  - OutputStream os = s.getOutputStream();
- Rock!
  - Build higher level things on the streams
ServerSocket

- Construct with the port number on which it should listen
  - `ServerSocket servsock = new ServerSocket(1234);`

- `Socket conn=servsock.accept()`
  - Will block until a client connects
  - Connection is then encapsulated in `conn`
  - Normally put in a loop to wait for next connection
Still too difficult?

- URL class
- Need to grab file from the web?
  - URL u = new URL("http://www.foo.com/a/b/c/d.txt");
  - InputStream is = u.openStream();
- Done!
- Much easier than C socket
- Also support ftp://, file://, https://