Course: COMP 303
Java SE review for assignment 1

Java Collections Framework

Design Patterns: DTO and Singleton

Java SE Networking

Multithreading

Java API: Utility Types

- Java gathers miscellanceous useful API into package **`java.util`**:
  - The collections framework → For aggregates of objects
  - Calendar, date and time data
  - Locales and resource bundles → For internationalization and localization
  - Random numbers
  - String tokenizer → For parsing strings
  - Regular expressions → For parsing strings — `java.util.regex`
  - Zip files — `java.util.zip`
  - and more

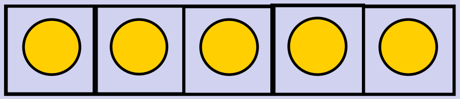
The Collection Framework

Objects are aggregates of objects

data structures

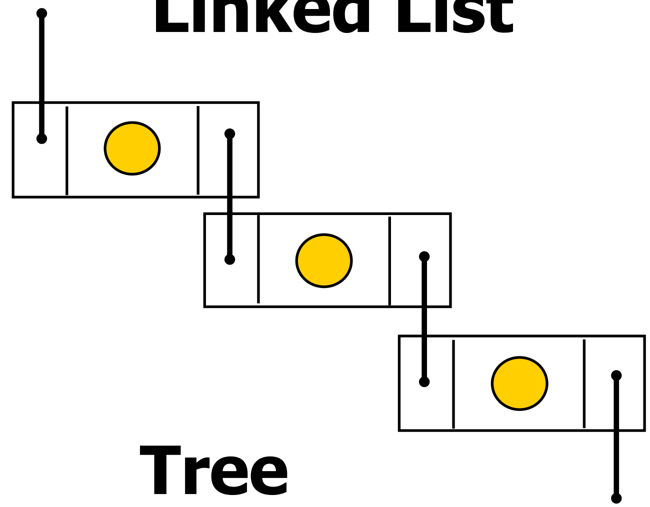
Collections represent data structures

Arrays and Strings are built into the Java language

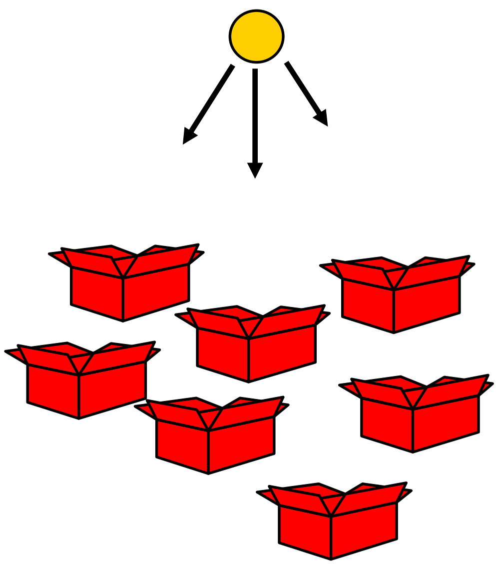


Aggregates but not members of the collection framework

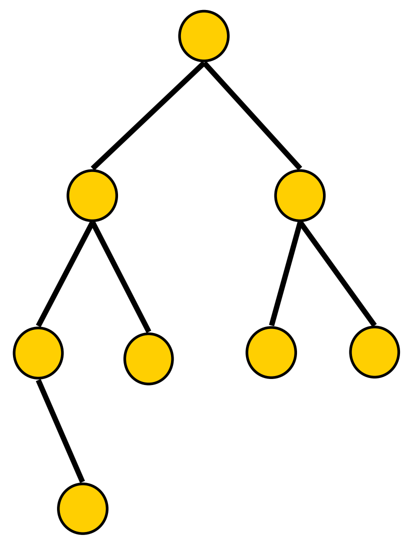
Linked List



Map - Hash



Tree



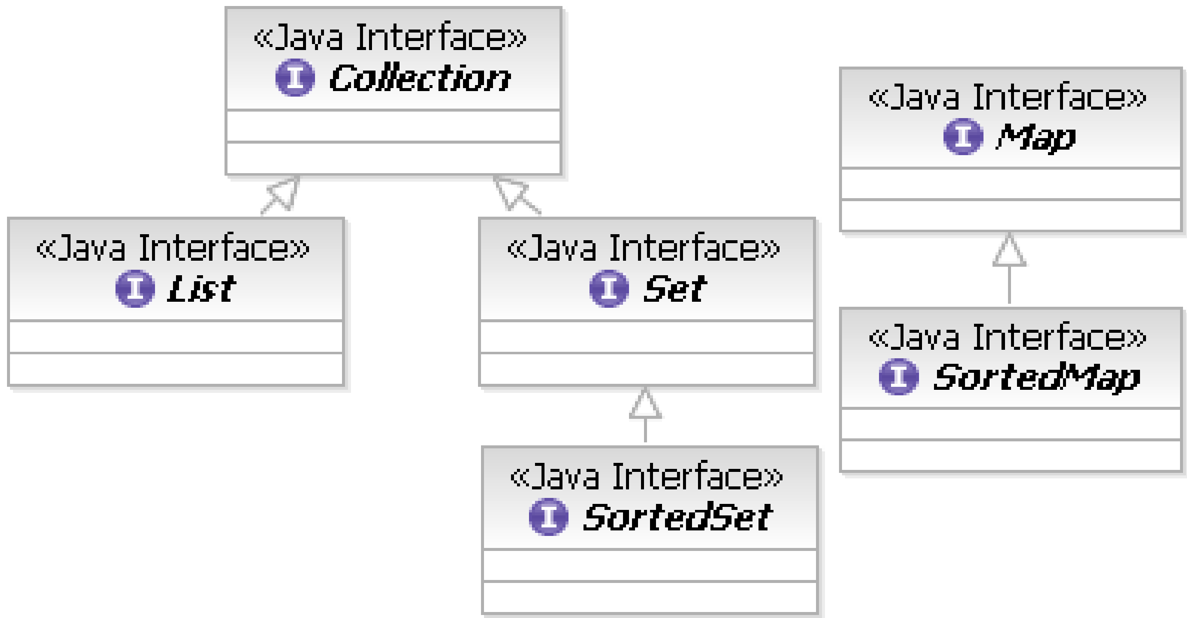
The Java Collections Framework

- The collections framework provides:
  - **Interfaces**
    - *Abstract data types representing collections*
    - *Allow collections to be manipulated independently of the details of their representation*
  - **Implementations**
    - *Concrete implementations of the collection interfaces*
    - *Reusable data structures*
  - **Algorithms**
    - *Methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces*
    - *Polymorphic same method can be used on many different implementations of the appropriate collections interface*
- **Online tutorial**
  - **http://docs.oracle.com/javase/tutorial/collections/**

Key interfaces in java.util

Collection Methods

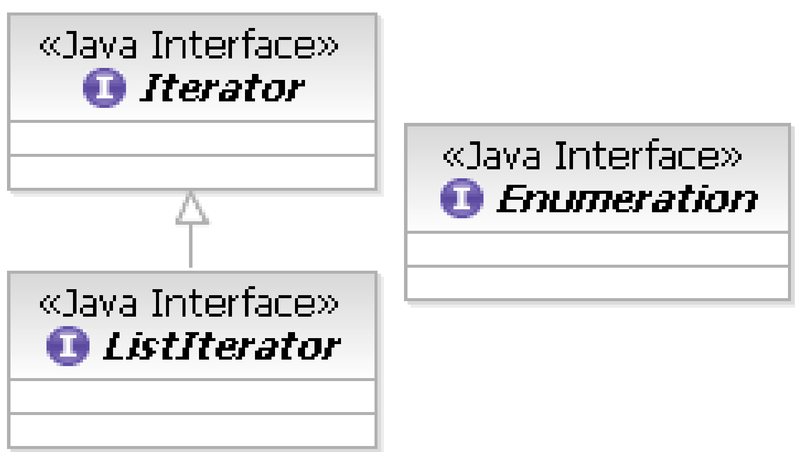
```
add()
clear()
contains()
isEmpty()
iterator()
remove()
size()
toArray()
```



Map Methods

```
clear()
containsKey()
containsValue()
entrySet()
get()
isEmpty()
put()
remove()
size()
values()
```

For iterating through a collection



For ranking/ordering elements

Concrete classes that implement the interfaces

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
INTERFACES	Set	HashSet LinkedHashSet EnumSet		TreeSet	LinkedHashSet	
	List		ArrayList		LinkedList	Vector Stack
	Map	HashMap		TreeMap	LinkedHashMap	HashTable Properties

Best practice: code to the interface rather than the implementation to promote portability

Some implementation choices

- **Set/Map**
  - **HashSet/HashMap**
    - *Very fast, no ordering*
  - **TreeSet/TreeMap**
    - *Maintains balanced tree, good for sorted iterations*
  - **Hashtable**
    - *Synchronized*
    - *Be sure to use **Map** interface*

- **List**
  - **ArrayList**
    - *Very fast*
  - **LinkedList**
    - *Good for volatile collection, or adding to the front of the list*
  - **Vector**
    - *Synchronized*
    - *Be sure to use **List** interface*

If none of the supplied implementations meet your requirements:

➢ Write your own by creating a class that implements on of the interfaces

✓ The collections framework is designed to be extensible

✗ Java programmers now rarely use arrays

Manipulating collections:

- Use generics specify type of contained object in a List

<type of contained objects>

```
List<String> colours = new ArrayList<String> ();
colours.add("red");
// add and remove more String objects
System.out.println( "There are " + colours.size() +
   "colours stored.");
```

- You can refer to ArrayList elements by position

```
System.out.println( "The first colour is" +
  colours.get(0));
// insert purple in as the 5th element
if ( colours.size() >= 4) {
    colours.add(4,"purple");
}
```

Iterating through a collection

- Special **for** syntax to iterate through set or list

```
for ( String colour : colours ) {
   if (colour.equals(“red”) {
        System.out.println(“Found red”);
   }
}
// Or, more simply
if (colours.contains(“red”) {
   System.out.println(“Found red”);
}
```

- Sometimes you need to use an Iterator or ListIterator

```
ListIterator li = colours.ListIterator();
while (li.hasNext()) {
    Colour colour = li.next();
     if (colour.equals(“red”) {
         System.out.println(“Found red”);
     }
}
```

Collections and generics: Maps

- Specifying type of contained object in a Map
  - *Elements implement interface Map.Entry<K,V>*

```
Map<String,Stock> stocks = new HashMap<String,Stock> ();
// add 100 shares of ABC at value = $123.25
Stock stock = new Stock(“ABC”, 123.25, 100);
// Elements in a map are (<key>,<value>)pairs
stocks.put(“ABC”, stock);
…
if (stocks.containsKey(“ABC” ) {
   System.out.println(“Quantity of ABC owned is” +
                    stocks.get(“ABC”).getQuantity)); }
// convert to ArrayList of Stocks
//          entryset() returns a Set<Map.Entry>
ArrayList<Stock> myStocks = new
                  ArrayList<Stock>(stocks.entrySet());
```

More collection operations

- Bulk operations on a Collection as a whole

```
boolean containsAll()
boolean addAll(Collection)
boolean
   retainAll(Collection)
boolean
   removeAll(Collection)
void clear()
```

- Bulk operations on a Maps as a whole

```
void putAll(Map)
void clear()
```

- Conversion to Arrays

```
Object[] o = c.toArray();

// c is ArrayList<String>
String[] a = c.toArray(new
   String[0];
```

- Converting Maps to Collections

```
Set<K> s = m.keySet();
Collection<V> c =
   m.values();
Set<Map.Entry<K,V>> es =
   m.entrySet();
```

Legacy Collection Classes

- Initially Java had just a few collection classes
  - They have been retrofitted into the collections framework
    - ***HashTable<K,V>***
    - ***Properties*** *extends HashTable where key and value are both Strings*
    - ***Vector*** *a List for which **Iterators** that are fail fast for use in multithreaded programs*
    - ***Stack** extends **Vector** for LIFO objects*
    - ***BitSet** all values are true or false*

Data Transfer Object

Value objects

Data Transfer Objects

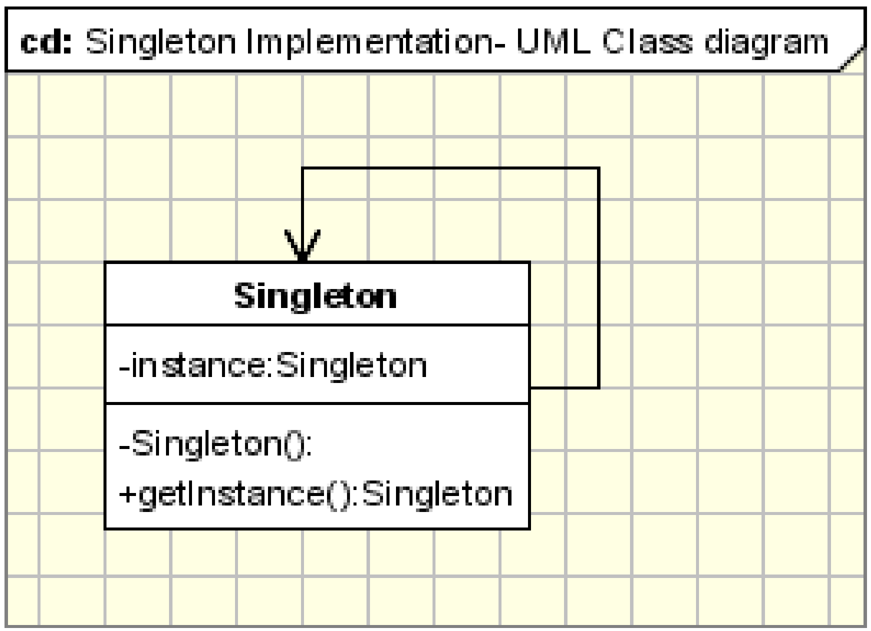
- Using DTO is as good programming practice
  - Gather data to pass between processes into one object
  - Return an object as the single return value of methods
    - *Methods can take one argument instead of a long list of arguments, and return an object of the same type as the arg*
  - Reduces use of multiple method calls to set different values
- DTO are usually implemented as a JavaBean
  - Properties with get and set methods
  - Minimal business logic beyond data validation in setters
  - Serializable for transmission over a wire

Unique objects

Singleton Design Pattern

Singleton design pattern

- Use singletons when:
  - You want only one instance of a class per JVM
  - To provide global access to the object



- Singleton is a creational design pattern
  - One of the original 23 patterns of the Gang of Four

Example Singleton Class

```
public class CatalogManager implements CourseCatalog {
      // instance of the class is a static field
      private static CatalogManager cm = null;
      private Map<String, Course> courses = null;
      // explicit private constructor
      private CatalogManager() {
         if ( courses == null ) {
               courses = new ConcurrentHashMap<String, Course> ();
      }
      // global way to access single instance
      public synchronized static CatalogManager getInstance() {
         if (cm == null ) {
               cm = new CatalogManager();
         }
         return cm;
      }
      // rest of class
```

Thread safely:
Create instance in synchronized block
Make all fields thread safe

Example class to use a singleton

```
public class CatalogClient {
     // no need to initialize singleton or store in a field
     public CatalogClient {
        super();
     }
     // method that uses singleton
     public Course lookUpCourse ( String courseCode)
                                        throws NoSuchCourseException {
       // access singleton with getInstance()
        CourseCatalog cc = CatalogManager.getInstance();
       // use singleton like any other object
       return cc.getCourse(courseCode);
     }
     // rest of class
 }
```

Multithreading

Reference: online Java SE tutorials

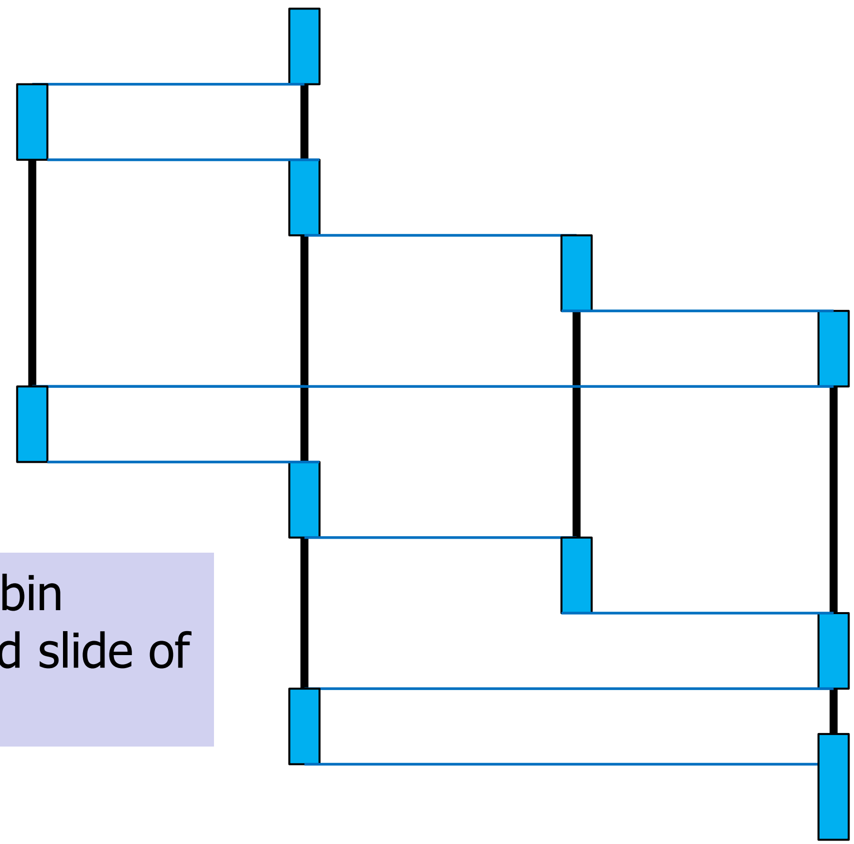
http://docs.oracle.com/javase/tutorial/essential/concurrency/

Threads

- Threads are lightweight processes
  - A process has its own execution environment
- Unlike most high level languages Java lets the program explicitly launch multiple threads:
  - The same code is executed on more than one thread

A running thread

A blocked thread



This diagram shows a simple round-robin scheme where each thread gets a fixed slide of time.

Sharing CPU cycles

- The great performance gain of multiple CPU is in ability to run more than one machine-level operation at once
- OS or JVM decides which thread to gets control
  - Algorithms vary with OS
  - Threads typically have priorities
- A thread gets then runs until it:
  - Runs out of time slice allocated by OS
  - Is blocked – usually because awaiting I/O or resource operation
  - Explicitly surrenders control
    - *In Java can be programmed using Thread API*
- JVM usually maintains several threads for all programs
  - Running application
  - Garbage collection
  - Refresh of graphical user interface display

Dangers with threads

- Data corruption
  - More than one thread updates the same variable
- Deadlock
  - All threads are stopped, waiting for their turn to run
- Starvation
  - Some threads don't get enough opportunity to run
  - Often causes by greedy threads
- Livelock
  - The JVM or OS spends so much time switching threads that thread code is starved

Launching a thread

- Classes whose object can run on a spawned thread must:
  - Implement **java.lang.Runnable** or
  - Extend **java.lang.Thread**

```
public class WorkerThread implements Runnable {
   public WorkerThread(…) {  … }
   @Override
   public void run() {
   // work done on multiple threads
   }
}
```

- Use class java.lang.Thread to manipulate Runnable objects

```
// main thread of a server in response to client request
   WorkerThread worker = new WorkerThread (…);
   Thread t = new Thread(worker);
   System.out.println("Thread started");
   t.start();
```

Manipulating threads

- Methods of the **java.lang.Thread** class typically called my main thread to control other threads
  - `start()` stop() is deprecated
  - `sleep()` suspend() is deprecated
  - `join()` resume() is deprecated
  - `getPriority()` `setPriority()`
  - `interrupt()`
  - `isAlive()`
  - `setDaemon()` `isDaemon()`
  - …
- Methods inherited from **java.lang.Object** used for communication between threads
  - `wait()` `notify()` `notifyall()`

// typical code for a Runnable class

```
public class WorkerThread implements Runnable{
// field to hold condition that indicates when to stop
   private volatile boolean stopFlag;
//
   public void run() {
      stopFlag = false;
      while ( !stopFlag) {
            // processing loop
            // do one logical unit of work
         try {
            // relinquish control
            wait();
         } catch ( InterruptedException ie ) {
            // woken up by another thread calling interupt()
         }
      } // end processing loop
      return;
   }
 // method called by controller to signal stop
   public void finish() {
       stopFlag=true;
   }
}
```

Atomic actions

- In multithreading, an atomic action is all at once
  - Not interrupted by code running on multiple threads
    - *Read and write for **most** reference variables and primitives except long and double*
    - *Read and write for all variables declared volatile*

- Expressions are not atomic, execution may be interleaved among threads
  - Example: x++
    - *One thread may increment x before or after another reads it*

Thread-safe variables

- When can there be conflict in concurrent access on more than one thread?
  - Value of variables
  - ✓ local variables
    - *Separate copies variable declared inside a method*
  - ✗ Instance variables
    - *Are specific to an object, but shared on threads*
    - ✓ *Can use an object of type ThreadLocal for thread safety*
  - ✗ Class variables
    - *Are shared by all objects and all threads*
    - ✓ *Can use an object of type ThreadLocal for thread safety*

Thread Synchronization

- Depends on concept of locks to prevent threads interfering with each other
- Achieve thread safety through "atomic actions"
  - Serialize access to methods or blocks of statements with keyword **synchronized** to apply implicit locks
- Atomic methods are not more efficient than synchronized methods, but harder to code
  - Declare variables to be volatile
  - Use classes that provide atomic methods such as those in **java.util.concurrent**
- Dangers:
  - Impact on performance
  - Possible deadlock where threads mutually block
  - Possible livelock where thread to busy interacting to make progress
  - Greedy threads starve others of resources

Java SE keywords for threads

- Serialize access to code with keyword **synchronized**

```
// generate account numbers starting at 1000000
private long LastAccoutNumber = 1000000;

public synchronized int getAccountNumber() {
       return ++lastAccountNumber;
}
```

  - Only one thread at a time can run synchronized code
  - Keyword synchronized can be applied to:
    - *class – all methods synchronized*
    - *method – put a lock on method*
    - *block of code – goal to synchronize a little as possible*
- Declare a variable **volatile** to tell the compiler it maybe changed by other threads
  - Prevent compiler optimizations that may loose changes made by other threads

Concurrency and collections

- Problem 1:
  - More than on thread updates an element in a collection at the same time
  - ➢ Data conflicts, unpredictable results
- Problem 2:
  - You use a collection such as ArrayList and visit elements using an Iterator or for loop such as:

    ```
    (for class c : classes ) { … }
    ```
  - Run the code on multiple threads that delete or add elements
  - ➢ The iterators can loose their place

➢ Solutions:
  - ✓ Use the collection in the package java.util.concurrent

    ```
    ConcurrentHashMap<KeyType, ValueType > myMap;
    ```
  - ✓ Generate synchronized collection for collection classes

    ```
    List sl = Collections.synchronizedList(
              new ArrayList<String>);
    ```

Collections and multithreading

- What happens when one thread inserts or deletes an element into a collection at the same time that another thread reads the collection?
  - With most collections results are not reliable
    - *The legacy collections are thread-safe*
  - Synchronizing collections can impact on performance
- The package **java.util.concurrent** contains versions of collections designed for scalable, multithreaded use:
  - ConcurrentHashMap
  - ConcurrentLinkedQueue
  - CopyOnWriteArraySet
  - …

Conclusions

- Best practise is to write every class as thought it might be run in a multithreaded environment
  - Consider thread safely of all variables and expressions and blocks
- Multithreading is often essential for
  - Performance
  - Conncurrent use
  - Real time processing
- Programming multithreading explicitly
  - Can be very tricky
  - Can produce bugs that are hard to reproduce and debug

- Java EE application servers provide a multithreaded environment
  - Developers writing components focus on business logic
  - The server handles multithreading

Networking

API in package **java.net**

Network programming with Java SE

- Package java.net API provide network abstractions:
  - Low level classes:
    - *Addresses identify host or socket endpoint (IP address)*
    - *Sockets establish communication link*
    - *Intrefaces to browse and query connections or endpoints*
  - Higher level classes:
    - *URI*
    - *URL*
    - *Connections*
- Lets you programming communications over
    - *TCP/IP (including URL) and datagram sockets*

Working with sockets

- Sockets are communications channels with numbered ports an both ends
- TCP/IP sockets are connection oriented
  - 2-way coversation over time
- Datagram sockets send packets one way over user datagram protocol (UDP)
  - Network address included in message
  - Faster, cheaper, no guarantee of delivery
  - Used by SNMP, games … where speed is more important that reliablility
  - Multicast socket for multicast groups

Using TCP/IP sockets

Two-way connection with handshaking

Packet validation by parity/CRC

Basis of all Internet communication

Creating client & server with (1 of 4) Socket and ServerSocket classes

- Client side defined by class MyClient

```
public class MyClient {
   static final int DEFAULT_PORT = 12345;
   // static final String hostIP = "10.28.39.18";
public static void main( String[] args ) {
     try {
       int port = DEFAULT_PORT;
       // optionally specify port as command-line argument
       if (args.length > 0 ) {
           port = Integer.parseInt(args[0]);
       }
      // InetAddress host = InetAddress.getByName(hostIP);
       InetAddress host = InetAddress.getLocalHost();
       // optionally specify host name as command-line argument
       if (args.length > 1 ) {
             host = InetAddress.getByName( args[1] );
      }
```

Creating client & server with Socket and ServerSocket classes (2 of 4)

- Client gets I/O streams from Socket

```
    Socket socket = new Socket( host, port );
    System.out.println("Demo client running");
 // Create PrintWriter: 2nd arg = true to autoflush buffer
    PrintWriter socketOut = new PrintWriter(
         socket.getOutputStream(), true );
    socketOut.println ("request for the server side");
 // read response from server
    BufferedReader socketIn = new BufferedReader( new
         InputStreamReader(socket.getInputStream() ) );
    String response = socketIn.readLine();
  } catch(UnknownHostException uhx) {
       uhx.printStackTrace();
  } catch(IOException iox ) {
       iox.printStackTrace();
  }
  // use response
  System.out.println(response);
}
```

Creating client & server with (3 of 4) Socket and ServerSocket classes

- Server side defined by class MyServer
  - Created at port provided by user or default

```
public class MyServer {
   private static int port;
   static final int DEFAULT_PORT = 12345;

 public static void main( String[] args ) {
      int port = DEFAULT_PORT;
      if (args.length > 0 ) {
         port = Integer.parseInt( args[0] );
      }
      // create a server socket bound to port
      ServerSocket ss = new ServerSocket( port );
      System.out.println("Demo server running");
```

Creating client & server with Socket and ServerSocket classes (4 of 4)

- Server gets socket from ServerSocket

```
   // process one request and respond
   try {
   // listen for TCP/IP connection from client on port
          Socket socket = ss.accept();
          BufferedReader in = new BufferedReader( new
             InputStreamReader( socket.getInputStream() ) );
          String str = in.readline();
   // send response back to client
          PrintWriter out = new PrintWriter(
             socket.getOutputStream(), true );
          String result = "response back to client";
  // echo result to server-side console
         out.println( "The result is " + result );
          socket.close();
    }  catch( IOException iox ) {
            iox.printStackTrace();
 }  }
```

Realistic servers

- Serve more than one client
- Can handle multiple requests at one time
  - Set up a listener to detect requests in server main()

```
ServerSocket listener = new ServerSocket(port);
Socket server;
```

  - Launch a new thread for each client
  - Loop until timeout or stopped by external operation

```
while ( true ) {
        socket = listener.accept();
        MyServerThread aServerThread = new
                MyServerThread(socket);
        Thread t = new Thread(aServerThread);
        t.start();
}
```

Creating a multithreaded server

1. Create a class that implements Runnable and overrides the run() method

```
public class MyServerThread implements Runnable {
   private Socket server;
   MyServerThread(Socket server) { this.server = server; }
   @Override
    public void run() {
       try {
          BufferedReader in = new BufferedReader( new
             InputStreamReader( server.getInputStream( ) ) );
          String str = in.readLine();
          PrintWriter out = new PrintWriter(
             server.getOutputStream(), true );
          String result = “response back to client”;
          out.println( "The result is " + result );
          server.close();
      }  catch( IOException iox ) {iox.printStackTrace(); }
   }
 }
```

Creating a multithreaded server

2. Launch a new thread for each client request

```
public class MultithreadedServer {
   private static final int DEFAULT_PORT = 12345;

   public static void main(String[] args) {
      int port = DEFAULT_PORT;
      if (args.length > 0) port = Integer.parseInt(args[0]);
      try {
         ServerSocket listener = new ServerSocket(port);
         Socket socket;
         while (true) {
            socket= listener.accept();
            MyServerThread aServer = new
                 MyServerThread(socket);
            Thread t = new Thread(aServer);
            t.start();
         }
      } catch (IOException ioe) { ioe.printStackTrace(); }
   }
}
```

Assignment 1

Use Eclipse and Java SE API to:

- Write classes and interfaces to create a simple client-server application
- Build a JavaBean to hold data
- Use the Java collections framework
- 2 design patterns:
  - *Singleton and Data Transfer Object*