Java Review

ECSE 321: Intro to Software Engineering
Electrical and Computer Engineering

McGill University

Winter 2009

Contents

1 Java Basics

1.1 Classes

A Simple Class

- **The** `Cube` **class is basically the same as a** `struct` **from C/C++**
- **Create a** `Cube` **object using** `new  Cube()`
- **Default constructor has no parameters and has the same name as the class.**
- **The programmer must remember to initialize each variable after creating the object.**

```
class  Cube{  int
   width    ;    int
   height   ;    int
   depth;
}
```

```
Cube c = new Cube( ) ; c
. width = 1;
c . height = 1; c
. depth = 1;
```

Overriding the Default Constructor

- **A *constructor* initializes an object upon creation.**
- **In the example below, variables are always assigned the same values.**
- **Still need to assign manually to change values.**

```
class  Cube{  int
   width    ;    int
   height   ;    int
   depth;

   public Cube(){
      width = 1;
      height = 1;
      depth = 1;
   }
}
```

```
Cube c = new Cube( ) ; c
. width = 2;
c . height = 2; c
. depth = 2;
```

Parameterized Constructors

- **Parameterized constructors make object creation easier:**
  - **Object creation and initialization is done using a single** `new` **statement.**
  - **Programmer need no longer initialize variables individually.**

```
class  Cube{
   int  width ;
   int   height ;
   int  depth;

   public  Cube( int     w,    int    h,    int    d){
      width  = w;
      height  =  h;
      depth  =  d;
   }
```

```
Cube  c1  = new  Cube(1 ,1 ,1);
Cube  c2  = new  Cube(2 ,2 ,2);
```

Methods

- **Member data should never be accessed directly.**
- **Use *get/set* methods to enforce *data encapsulation*.**

```
class  Cube{
   private    int    width    ;
   private    int    height    ;
   private int depth;

   public  Cube( int  w,   int  h,   int  d)  {
      width  = w;
      height  =  h;
      depth  =  d;
   }

   public void setWidth( int w) { width =
      w;
   }

   public int getWidth () { return
      width ;
   }
}
```

```
Cube c = new Cube(1 ,1 ,1); c .
setWidth (2);
int  w =  c . getWidth ( ) ;
```

Overloading

- **Can define two or more methods within the same class that share the same name.**
- **Parameters must be different in order to distinguish between methods.**

```
class  Cube{
   . . .

   public  Cube  ( int  w,   int  h,   int  d)  {
       . . .
   }

   public  Cube( float  w,   float  h,   float  d)  {
      this (( integer )  w,   ( integer )  h,   ( integer )  d) ;
   }

   public  void  setWidth( int  w)  {
      width  = w;
   }

   public void setWidth( double w) {
      setWidth (( integer ) w) ;
   }
}
```

EchoArgs.java

- **EchoArgs simply echoes command-line parameters.**

```
public    class  EchoArgs  {

   public  EchoArgs( String []   str )  {
      for ( int   i  =  0;   i  <  str . length ;          i++)  {
         System . out . println ( str [ i ] ) ;
      }
   }

   public   static   void  main( String []           args)  {
      i f (args . length  ==  0)  {
         System . out . println ( "no  args  to  echo . . . " ) ;
      }   else  {
         new  EchoArgs(args ) ;
      }
   }
}
```

- **Compiling and Running EchoArgs:**

```
$  javac  EchoArgs . java
$  java    EchoArgs
no  args  to  echo . . .
$  java  EchoArgs  one  two  three
one
two
three
```

1.2 Static and Final

The `Static` Modifier

- **Data: Same data is used for all the instances (objects) of some Class.**
- **Method: Can be called without an instance and can only access static data.**
- **Initialization Block: A block of code that is executed when the class is first loaded.**

```
class  StaticExample  {
   static   int  INSTANCES =  0;
   static{
      System . out . println ( " Static   Initializer "};
   }
   public  StaticExample ()  {
      INSTANCES++;
   }
   public static int getInstances () { return
      INSTANCES;
   }
}
```

- `static` **variables/blocks are initialized/executed in the same order that they appear in the code.**

The `Final` Modifier

```
public   final   class  FinalClass  {
       static final int STATIC_CONSTANT = 1; final
       int CONSTANT = 2;

       final   void  finalMethod ()  {}
}

public class BrokenClass
       extends FinalClass {

       void finalMethod () {
             CONSTANT = 0;
       }
}
```

- `final` **member data cannot be changed (constant).**
- `final` **method cannot be overridden by a subclass.**
- `final` **class cannot be extended.**
- **Final can be used to prevent errors.**
- `BrokenClass` **will not compile because:**
  - **It extends a final class.**
  - **It overrides a final method.**
  - **It assigns a new value to a final variable.**

Singly Linked List

```
class  Node{
   public Object data = null ; public
   Node next = null ;
}
```

```
public class SinglyLinkedList{ static final
   int MAX_SIZE = 10; final Node HEAD =
   new Node( ) ; Node tail = null ;

    int   size  =  0;

   public  void  addNode(Object        data)    {
       if ( size  == MAX_SIZE}
          return ;
       if ( size  ==  0)  {
         HEAD. data  =  data ;
          tail  = HEAD;
       }  else   {
          Node  n  = new  Node( ) ;
          n. data  =  data ;
          tail . next  =  n;
          tail  =  n;
       }
      size++;
   }
}
```

- `MAX¯SIZE` **shared by all** `SinglyLinkedList` **objects.**
- `HEAD` **declared as final to prevent us from changing it by mistake.**

1.4 Access Control

Access Control

- `public`
  - ***class:*** **access granted to everyone**
  - ***member function/data*****: Can be called/modified by other classes.**
- `protected` **can be called/modified from derived classes only.**
- `private` **can be called/modified only from the current class**
- **By *default*, when no access specifier is used, the member/class can be called/modified/instantiated only from *within* the same package.**

1.5 Packages

Packages

- **A package physically and logically groups classes together**
- **Avoids naming conflicts**

- **Control access to classes**
  - **Unrestricted access between classes of the same package *(* public).**
  - **Restricted access for classes outside the package *(* default).**
- **Place a package statement at the top of the source file in which the class or the interface is defined.**
- **Refer to a member by its *qualified* name, ie.** `java.util.LinkeList`
- **Importing classes:**
  - **Import statements go after package statement.**
  - **A single class:** `import  java.util.LinkedList;`
  - **All classes from a package:** `import  java.util.*;`

```
package  my. utils ;

public    class          UtilClass{
}

class   HelperClass{
}
```

```
package  my. ds;

public    class        LinkedList{
}
```

```
import  my. utils . * ;
import  java . util . LinkedList ;

public  class     Program  {
       UtilClass      util  = new  UtilClass ( ) ;
            HelperClass  help   = new  HelperClass ( ) ;
          my. ds . LinkedList  l i s t  = my. ds . LinkedList ( ) ;
}
```

- **Will not work because** `HelperClass` **is not public.**
- **Using *qualified* name avoids name conflicts with** `LinkedList` **class;**

1.6 Mutability and Immutability

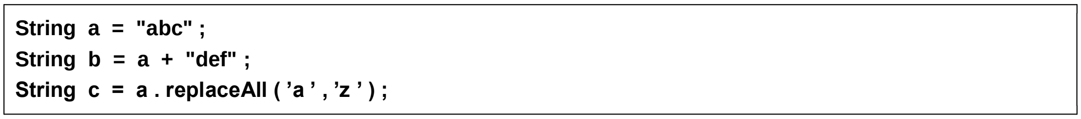
Mutability

- **An object is mutable if it has methods which can change its state.**
- **The** `StringBuffer` **class can be modified dynamically**

```
StringBuffer str = new StringBuffer ( "abc" ) ; str
.append( "def" ) ;
```

Immutability

- **An object is immutable if it cannot be changed.**
- **The** `String` **class is immutable since it doesn’t have any methods that let you change it’s state.**
- **What about the** `replace` **methods?**
  - **they return a new** `String` **object.**



```
String a = "abc" ;
String b = a + "def" ;
String c = a . replaceAll ( ’a ’ , ’z ’ ) ;
```

2 Objects

2.1 The Object Class

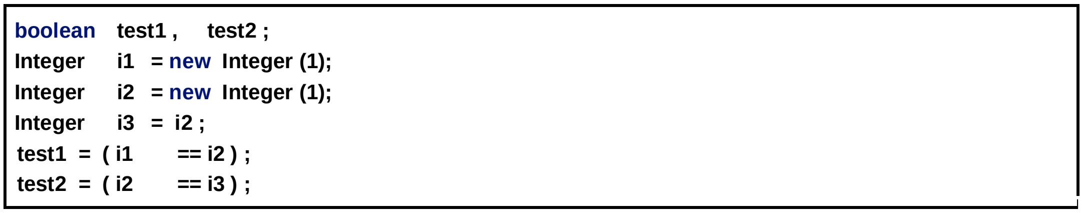
The `Object` Class

- **Every Class in Java extends** `java.lang.Object`**.**
- **Provides methods that are common to *all* objects.**
- **Some of the methods defined by the** `Object` **class are:**
  - `Object clone()`**: Creates a new object that is the same**
  - `boolean equals(Object o)`**: Determines whether one object is equal to another**
  - `void finalize()`**: Called before an object is destroyed**
  - `int hashCode()` **Returns the hash code associated with an object**
  - `String toString()`**: Returns a string that describes the object**

2.3 Equality

Equality Operator

- **The equality operator** `==` **returns true if and only if both its operands have the same value.**
- **Can be used to compare primitive types**
- **Only compares the values of reference variables, *not the referenced objects*.**



```
boolean   test1 ,   test2 ;
Integer   i1  = new  Integer (1);
Integer   i2  = new  Integer (1);
Integer   i3  =  i2 ;
test1  =  ( i1      == i2 ) ;
test2  =  ( i2      == i3 ) ;
```

- **test1 equals** `false`
- **test2 equals** `true`

Object Equality I

- **To compare between two objects the** `boolean equals(Object o)` **method is used:**
  - **Compares the *contents* of two objects and returns true if the objects are equivalent.**
  - **Default implementation compares using the equality operator.**
  - **Override this method to provide your own implementation.**
  - `hashCode()` **must produce the same result for two objects that are found to be equal via** `equals(Object o)`

```
boolean test1 , test2 ;
String s1 = new String ( "abc" ) ;
String s2 = new String ( "abc" ) ;
String s3 = new String ( "def" ) ; test1
= s1 . equals(s2 ) ;
test2 = s2 . equals(s3 ) ;
```

- **test1 equals** `true`
- **test2 equals** `false`

Object Equality II

- **You may need to overload** `equals` **for custom classes:**

```
public class Name {
      String firstName ;
      String lastName;

      public   boolean equals(Object o)        {
            i f  ( ! ( o instanceof Name))
                  return = false ;
            Name n = (Name) o;
            return firstName . equals(n. firstName) && lastName . equals(n. lastName) ;
      }
}
```

`equals()` vs. ==

- **It is important to remember that the** `equals()` **method compares the *contents* of an object while** `==` **compares two object references for equality.**

```
boolean test1 , test2 ;
String s1 = new String ( "abc" ) ;
String s2 = new String ( "abc" ) ;
test1 = s1 . equals(s2 ) ;
test2 = (s1 == s2 ) ;
```

- **test1 equals** `true`
- **test2 equals** `false`

2.3 Cloning Objects

Cloning Objects I

- **The `clone()` method generates a duplicate copy of the object on which it is called.**
- **Only classes that implement the `Cloneable` interface can be cloned, otherwise a `CloneNotSupportedException` will be thrown.**
- **The constructor for the object being cloned is *not* called; a clone is simple *an exact copy of the original*.**
- **Cloning can be dangerous**
  - **If an object being cloned contains a reference to an object, the reference is copied, resulting in original and cloned objects referencing the same object.**
- **`clone()` is *protected* inside `Object`.**

Cloning Objects II

- **In general, you should not implement `Cloneable` for any class without good reason.**
- **Safer to write a `copy` method yourself which creates new objects using constructors.**

```
class Test implements Cloneable{ int a;

   double  b;

   public Object clone(){ try{

         return  super . clone ( ) ;
      } catch (CloneNotSupportedException e) { e .
         printStackTrace (System . out ) ;
      }
   }
}
```

```
class   Copy{
   int   a;
   double  b;

   public  Test( int       x , double   y)    {
      a  =  x;
      b  =  y;
   }

   public  Test       copy() {
      return  new  Test(a,b) ;
   }
}
```

3 Object Oriented Programming

3.1 Encapsulation

Encapsulation

- ***Encapsulation*** **is the mechanism that binds together code and the data it manipulates.**
- **Keeps code and data safe from outside forces.**
- **Access to code and data is strictly controlled by a well defined interface**
  - **Prefer member data to be** `protected` **or** `private`
  - **Access member data via *get* and *set* methods**
- **Implementation details are kept hidden behind the interface; *encapsulate complexity***

3.3 Inheritance

Inheritance

- ***Inheritance*** **is the process by which one object acquires the properties of another object.**
- **Allows the definition of hierarchies.**
- **Enables *code reuse*.**
- **Java *does not allow multiple inheritance*:**
  - **Implementing multiple interfaces is allowed.**

Equality and Inheritance

- **Inheritance can cause problems with equality.**
- **Why can a subclass require a new implementation of** `equals`**?**
  - **New fields in the subclass are not taken into account by the superclass.**

```
public class FullName extends Name{
    String middleName;

    public boolean equals(Object o) {
        if (!(o instanceof FullName))
            return false;
        FullName n = (FullName) o;
        return super.equals(o) && middleName.equals(n.middleName);
    }
}
```

Abstract Classes

- **An abstract member function does not have an implementation.**
- **An abstract class cannot be instantiated.**
- **If class is abstract if one or more methods are declared abstract.**

```
public abstract class Shape{ public
        abstract void draw( ) ;
}
```

```
public class Circle extends Shape{ public
        void draw() {
                // draw a circle
        }
}
```

3.3 Interface

Interface

- **Defines a protocol of communication between two objects**
- **Contains declarations but no implementations**
- **All methods are public**
- **All fields are public, static and final (constants).**
- **Java's compensation for removing multiple inheritance. You can *implement* as many interfaces as you want.**

```
interface Producer {
        Object produce ( ) ;
}

interface Comsumer {
        void consume(Object o}
}
```

```
public class ProducerConsumer implements
        Producer , Consumer {

        public Object produce() {
                return (Object) new String ( "abc" ) ;
        }
        public void consume(Object o) { System . out ,
                println (o . toString ( ) ) ;
        }
}
```

4 Exceptions

4.1 Exceptions

Exceptions

- **An *exception* is an abnormal condition that arises at run time; a *runtime error*.**
- **When an error occurs, an `Exception` object is *thrown*.**
- **A *thrown* exception must be *caught* in order to handle it.**
- **Exceptions can be produced by:**
    - **the Java runtime system**
    - **manually generated exceptions**

Keywords

- **There are five keywords relating to exceptions:**
    - `try` **blocks contain code being monitored for errors.**
    - `catch` **contains the code that will handle the exception**
    - `throw` **creates an** `Exception` **object.**
    - `throws` **denotes the exception types a method can generate.**
    - `finally` **contains code that will be executed before a try block ends.**

Exception Types

- **All exception types are subclasses of** `Throwable`**:**
    - `Exception`**: error conditions that the user program should handle.**
        - `RuntimeException`**: automatically defined error conditions like divide-by-zero.**
    - `Error`**: error conditions that a program is not expected to handle.**

4.3 Basic Use

Uncaught Exceptions

- **What happens if we don't handle errors?**
    - **The default handler will print a stack trace and terminate the program.**

```
public  class  Exceptions{

   public static void main( String [] args) { Object o =
      null ;
     System . out . println (o . toString ( ) ) ;
   }
}
```

```
$ java Exceptions Exception in
thread "main"
   java . lang . NullPointerException
   at  Exceptions .main(Exceptions . java :5)
```

Using `try` and `catch`

- **Handle the exception yourself:**
  - **Fix the error.**
  - **Prevent program termination**
- `try` **and** `catch` **form a unit.**

```
public   class  Exceptions{

   public  static  void  main( String []   args)            {
      Object  o  =  null ;
      try{
         System . out . println (o . toString ( ) ) ;
      }  catch  ( NullPointerException  e)  {
        System . out . println ( "caught  nullptr " ) ;
      }
      System . out . println ( " s t i l l   going . . . " ) ;
   }
}
```

```
$  java    Exceptions
caught    nullptr
 s t i l l  going . . .
```

Multiple `catch` clauses

- **If a piece of code can generate multiple exception types, use multiple** `catch` **clauses to deal with each.**

- **Each** `catch` **clause is inspected in order until a match is found.**

```
public  class  Exceptions{

   public static void main( String [] args) { Object []
      array = new Object [5];
      try{
         array [10]. toString ( ) ;
      }  catch  ( NullPointerException  e)  {
      } catch (ArrayIndexOutOfBoundsException e) { System .
         out . println ( "caught exception" ) ;
      }
      System . out . println ( " s t i l l   going . . . " ) ;
   }
}
```

```
$ java Exceptions
caught exception s t i
l l going . . .
```

Nested `try` Statements

- **We can nest try statements.**
- **If an inner `try` statement doesn't have a handler, the outer `try` statement is inspected until a match is found.**

```
public  class  Exceptions{

   public static void main( String [] args) { Object []
      array = new Object [5];
      try{
         try{
            array [10]. toString ( ) ;
         } catch ( NullPointerException e) { System .
            out . println ( " inside" ) ;
         }
      } catch (ArrayIndexOutOfBoundsException e) { System .
         out . println ( "outside" ) ;
      }
      System . out . println ( " still  going . . . " ) ;
   }
}
```

```
$ java Exceptions
outside
still  going . . .
```

4.3 Throw

`throw`

- **We can `throw` exceptions explicitly**
- **We can create objects of type `Throwable`.**
- **We can use `throw` to create a new exception object or to re-throw a caught exception.**

```
public  class  Exceptions{

   public static void main( String [] args) { Object []
      array = new Object [5];
      try{
         try{
            array [10]. toString ( ) ; }
         catch (Exception e) {
            throw  new  Exception( "my own  message" ) ;
         }
      }  catch  (Exception  e)  {
         System . out . println (e . toString ( ) ) ;
      }
      System . out . println ( " still  going . . . " ) ;
   }
}
```

```
$ java Exceptions java .
lang . Exception :
   my own message
still going . . .
```

Tip: use Exception.PrintStatckTrace

- **Printing the stack-trace when you catch an exception will help you find your error.**

```
public class Exceptions{

   public static void oops( int x) {
      try {
         if (x == 0) {
            throw new Exception( "oops" ) ;
         } else {
            oops(−−x ) ;
         }
      } catch (Exception e) {
        e. printStackTrace ( ) ;
      }
   }

   public static void main( String [] args) { oops(5);

   }
}
```

```
$ java Exceptions
java . lang . Exception : oops
   at Exceptions . oops(Exceptions . java :6) at
   Exceptions . oops(Exceptions . java :8) at
   Exceptions . oops(Exceptions . java :8) at
   Exceptions . oops(Exceptions . java :8) at
   Exceptions . oops(Exceptions . java :8) at
   Exceptions . oops(Exceptions . java :8) at
   Exceptions .main(Exceptions . java:16)
```

4.4 Throws

`throws`

- **If a method generates an exception that it doesn't handle, it must let the calling method know via the `throws` clause.**
- **List all possible exceptions after the `throws` clause.**
- **Caller is responsible for handling the exception.**

```
public class Exceptions{


   public static void oops() throws
      NullPointerException {
        throw  new  NullPointerException ( ) ;
    }

    public    static    void  main( String []      args)     {
       try{
          oops ( ) ;
       } catch  (Exception  e)  {
          e . printStackTrace ( ) ;
       }
    }
}
```

```
$ java  Exceptions
java . lang . NullPointerException
   at Exceptions . oops(Exceptions . java :4) at
   Exceptions .main(Exceptions . java :9)
```

4.5 Finally

finally

- When exceptions are thrown, the execution flow of the program becomes non-linear.
- An exception can cause a method to return abruptly; we may want to do some cleanup first:
  - Close open files
  - Free shared resources in the case of multi-threading
- `finally`:
  - designates a block of code which is to be executed following a try/catch block.
  - will execute whether or not an exception is thrown.
  - will execute whether or not a `catch` statement matches the exception.
  - will execute just before a method returns

Example

```
public  class  Exceptions{

   public static void oops() throws
      NullPointerException { try {

         throw new NullPointerException ( ) ; }
      finally {
         System . out . println ( "Cleaning  up . . . " ) ;
      }
   }

   public   static   void  main( String []  args)  {
      try{
         oops ( ) ;
      } catch  (Exception  e)  {
         e. printStackTrace  ( )  ; }
      finally {
            System . out . println ( "Exiting" ) ;
        }
    }
}
```

```
$ java Exceptions
Cleaning up . . .
java . lang . NullPointerException
            at Exceptions . oops(Exceptions . java :5) at
            Exceptions .main(Exceptions . java:13)
Exiting
```

5 Collections

5.1 Collections Framework

Collections Framework

- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- All collections frameworks contain three things:
  - *Interfaces* allow collections to be manipulated independently of the details of their represen-tation.
  - Concrete *Implementations* of the collection interfaces.
  - *Algorithms* like searching and sorting on objects that implement collection interfaces.
- Algorithms represent reusable functionality; they can be applied to different implementations of the collection interfaces.

Why Use The Collections Framework?

- Reduces programming effort by providing useful data structures and algorithms.
- Increases program speed and quality: The collections framework does this primarily by providing high-performance, high-quality implementations of useful data structures and algorithms.
- Reduces the effort to learn and design use new APIs.
- Enables software reuse

5.3 Collections Interfaces

Interfaces

- The `Collection` interface is the root of the collection hierarchy.
  - A `Set` is a collection that cannot contain duplicate elements (`HashSet, TreeSet`) .
  - A `List` is an ordered collection and can contain duplicate elements (`ArrayList,LinkedList`).
  - A `Map` is an object that maps keys to values and cannot contain duplicate key (`HashMap, Hashtable`).
- For more info, visit http://java.sun.com/docs/books/tutorial/collections/index.html

`Collection` Interface

```
public interface Collection { int size (
        ) ;
        boolean  isEmpty ( ) ;
        boolean contains (Object element ) ;
        boolean add(Object element ) ; boolean
        remove(Object element ) ; Iterator iterator
        ( ) ;

        boolean containsAll ( Collection c ) ;
        boolean addAll ( Collection c ) ; boolean
        removeAll( Collection c ) ; boolean
        retainAll ( Collection c ) ; void clear ( ) ;


        Object []    toArray ( ) ;
        Object []    toArray(Object      a [ ] ) ;
}
```

5.3 Implementations

`ArrayList` **or** `LinkedList`

- `ArrayList` **offers constant time positional access and is fast.**
- `LinkedList` **If you frequently add elements to the beginning of the List, or iterate over the List deleting elements from its interior**
- `ArrayList` **is much faster, use it instead of** `LinkedList` **unless you really need it's added features.**
- **The** `Vector` **class has been kept for backwards compatibility and should be avoided.**

`HashSet/Map` or `TreeSet/Map`

- `HashSet/Map` **is much faster (constant time vs. log time for most operations), but offers no order-ing guarantees.**
- `TreeSet/Map` **If you need to use the operations in the** `SortedSet`**, or in-order iteration is important to you.**
- **Mostly use** `HashSet` **and** `HashMap`

`Collection` Example

```
import  java . util . * ;

class  CollectionExample  {

        public   static      void  main( String []   args)       {
               ArrayList      al  = new  ArrayList ( ) ;
               al . add( "zero" ) ;
               al  .  add(  "one"  )  ;
               al . add( "two" ) ;
               System . out . println (
            al . get (1) . toString ( ) ) ;

             HashMap hm = new HashMap( ) ;
             hm. put( "a" ,  new  Integer (1));
             hm. put( "b" ,  new  Integer (2));
             System . out . println (
             hm. get( "a" ) . toString ( ) ) ;
        }
}
```

```
 $ java    CollectionExample
one
1
```

5.4 Iterators

Interator Interface

- **An iterator allows us to access the elements of a collection.**
- **Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.**

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

Using `ListIterator`

```
import  java . util . * ;

public  class  LinkedListExample  {
   public   static      void  main( String []   args)        {
      LinkedList     list  = new  LinkedList ( ) ;
      list . add( "one" ) ;
      list . add(new  Integer (1));
      list . add(new  LinkedList ( ) ) ;

      System . out . println ( " list . toString ( ) :        "
         +  list . toString ( ) ) ;
      ListIterator   it  =  list . listIterator ( ) ;
      while ( it . hasNext ( ) )  {
         Object  o  =  it . next ( ) ;
         System . out . println ( "o . toString ( ) :   "
            +  o . toString ( ) ) ;
      }
   }
}
```

```
$ java  LinkedListExample
list . toString ( ) :   [one,   1,   [ ] ]
```

```
o . toString ( ) : one o
. toString ( ) : 1 o .
toString ( ) : []
```

6 Java IO

6.1 Streams

Streams

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes data.
- All streams behave in the same manner, regardless of the actual physical device.
  - *the same I/O classes and methods can be applied to any type of device.*
- The stream classes are in the `java.io` package.

6.3 Byte Streams

Byte Streams I

- The byte stream classes provide facilities for handling byte-oriented I/O.
- Read/Write 8-bit bytes
- Based on two abstract classes:
  - `InputStream`
  - `OutputStream`
- Can improve performance by using `BufferedInputStream` and `BufferedOutputStream`
  - May need to call `flush()` to cause data that is in a buffer to be written.

Byte Streams II

- Can read/write binary data to and from files using `FileInputStream` and `FileOutputStream`

```
import java . io . * ;

class CopyFile{
   public copyFile ( String in ,  String out) throws IOException{
      File  inputFile  = new File ( in ) ;
      File  outputFile  = new File (out ) ;

      FileInputStream ins = new FileInputStream( inputFile ) ;
      BufferedOutputStream bos = new BufferedOutputStream(
         new FileOutputStream( outputFile ) ) ; int
      c;

      while (( c = ins . read ( ) ) != −1) { bos .
         write (c ) ;
      }
      ins . close ( ) ;
      bos . flush ( ) ;
      bos . close ( ) ;
   }
}
```

6.3 Character Streams

Character Streams I

- The byte stream classes provide functionality to handle any type of I/O.
- How can we easily handle character data?
  - We can use the `Reader` and `Writer` abstract classes.
- Can improve performance by using `BufferedReader` and `BufferedWriter`
  - `BufferedReader.readLine()` method reads a line of text.
  - Remember to call `flush()`

Character Streams II

- Can read/write character data to and from files using `FileReader` and `FileWriter`

```
import  java . io . * ;

class  ReadLines{
   public readFile ( String in ) throws IOException{ File
      inputFile = new File ( in ) ;

      BufferedReader br = new BufferedReader( new
         FileReader( inputFile ) ) ;

      String   line ;
      while (( line = br . readLine ( ) ) != null ) { System .
         out . println ( line ) ;
      }
      br . close ( ) ;
   }
}
```

Reading Console Input

- Console input is read from `System.in`
- Use `BufferedReader` to get a character based stream.

```
import  java . io . * ;

class  ReadConsole  {

public  readFromConsole()
```

```
   throws  IOException  {

   char  c;
    BufferedReader  br  = new  BufferedReader( new
   InputStreamReader(System . in ) ) ; System . out .
            println ( "Enter  characters : " ) ;
   do  {
      c = ( char ) br . read ( ) ;
      System . out . println (c ) ;
   }  while ( true ) ;
}
```

6.4 Serialization

Serialization

- ***Serialization*** **is the process of writing the state of an object to a byte stream.**
- **Can save the sate of a program to persistent storage and restore objects at a later time.**
- **Can send objects back and forth over a network.**
- **If an object to be serialized contains references to other objects, these objects must also be serialized.**
- **Only objects that implement the** `Serializable` **interface can be saved and restored by serializa-tion.**
  - **Serializable interface defines no members, it is simply used to indicate that a class is serial-izable**
  - `transient` **and** `static` **variables are not saved.**

ObjectOutput and ObjectInput

- `ObjectOutputStream` **extends the** `OutputStream` **class and implements the** `ObjectOutput` **interface.**
- `ObjectInputStream` **extends the** `InputStream` **class and implements the** `ObjectInput` **interface.**
- **We can use the respective stream classes to easily serialize/deserialize objects.**

Serialization Example

```
import  java . io . * ;

public class MySerial implements Serializable { String data
   ;

   public MySerial( String s) { data =
      s ;
   }

   public String toString () { return
      data ;
   }
   public  static  void  main( String []  args)  throws          Exception  {
      FileOutputStream      out  = new  FileOutputStream( "myserial . bin" ) ;
      ObjectOutputStream  outs  = new  ObjectOutputStream(out ) ;
      MySerial  obj  = new  MySerial( " I 've  been  serialized ! " ) ;
      outs . writeObject ( obj ) ;
      outs . flush ( ) ;
      outs . close ( ) ;

      FileInputStream      in  = new  FileInputStream( "myserial . bin" ) ;
      ObjectInputStream  ins  = new  ObjectInputStream( in ) ;
      obj  =  (MySerial)       ins . readObject ( ) ;
      System . out . println ( obj . toString ( ) ) ;
      ins . close ( ) ;
   }
}
```

- Compiling and running MySerial.java:

```
$  javac  MySerial . java
$  java  MySerial
I ' ve  been  serialized !
```

7 Assertions

7.1 Assertions

Assertions

- Each assertion contains a boolean expression that should be true when the assertion executes.
- If an assertion evaluates to `false`, the system will throw an error.
- Using assertions is one of the quickest and most effective ways to detect and correct bugs.
- Remember that assertions can be enabled and disabled.

Syntax

- `assert Expression1;` where Expression1 is a boolean expression. When the system runs the assertion, it evaluates Expression1 and if it is false throws an AssertionError with no detail mes-sage.
- `assert Expression1 : Expression2;` where Expression1 is a boolean expression and Expression2 is an expression that has a value.
- Use the second form to provide a detailed message for constructing the `AssertionError` object.

When not to use assertions

- Do not use assertions for argument checking in public methods.
- Do not use assertions to do any work that your application requires for correct operation.

7.2 Example

Simple Example

- **Use assertions whenever you've made assumptions about:**
  - **the legal values of a variable.**
  - **flow control.**
- `switch` **statement with no** `default` **case assumes that one of the cases is always executed.**
- **Use the** `default` **case to test our assumptions**

```
public class  Assert{
  public  static  void  main( String []  args) {
    int  x  =  2;
    switch (x)  {
      case  0:
        break ;
      case  1:
         break ;
      default :
        assert  false  :  "value  of  x  is :  "  +  x;
          break ;
      }
  }
}
```

```
$  javac  −source  1.4  Assert . java
$  java  Assert
$  java  −ea  Assert
Exception in thread "main" java .
      lang . AssertionError : value of
      x is : 2
      at  Assert .main( Assert . java:13)
```

7.3 Preconditions and Postconditions

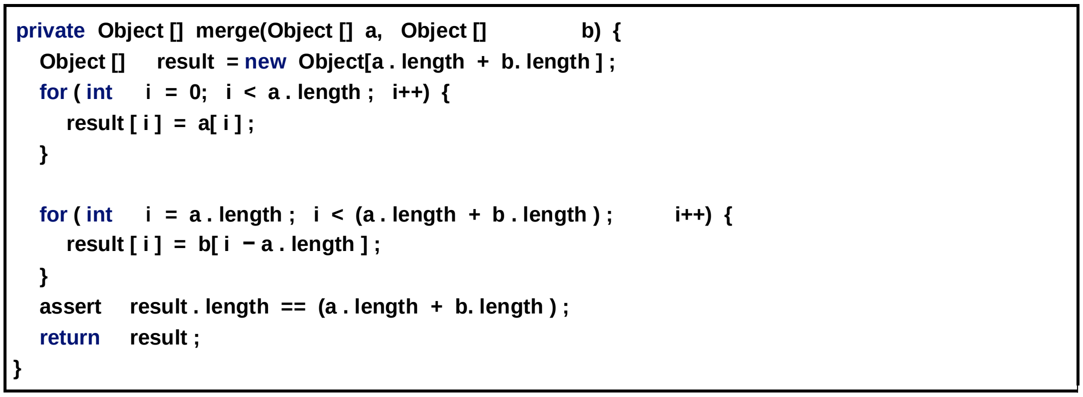
Preconditions

- ***Preconditions*** **must be true when a method is invoked.**
- **Do not use assertions to check the parameters of a *public* method, use *exceptions* instead.**
- **Use an assertion to test a *nonpublic* method's precondition that should always be true.**

```
private void setUpperBound( int x) { assert
  (x >= 0) : "upper bound
    must  be  positive " ;
}
```

Postconditions

- ***Postconditions*** **must be true after a method completes successfully.**



```
private  Object []  merge(Object []  a,   Object []            b) {
   Object []     result  = new  Object[a . length  +  b. length ] ;
   for ( int     i  =  0;   i  <  a . length ;   i++) {
      result [ i ]  =  a[ i ] ;
   }

   for ( int     i  =  a . length ;   i  <  (a . length  +  b . length ) ;        i++) {
      result [ i ]  =  b[ i  − a . length ] ;
   }
   assert    result . length  ==  (a . length  +  b. length ) ;
   return    result ;
}
```

7.4 Final Thoughts

Final Thoughts on Assertions

- **You can use them in place of** `print` **statements.**
- **They are similar to exceptions.**
- **Only available as of JDK 1.4.**
- **Use** `java -source 1.4` **to compile.**
- **Use** `java -ea` **to enable them.**
- **Prefer Exceptions to Assertions**

8 Log4j

8.1 Logging for Java

What's wrong with `System.out.println()`?

- **You need to recompile your program in order to add/remove print statements. – not practical for large applications.**
- **No good way to control verbosity unless you write your own logging framework.**
- **Output format of print statements is often inconsistent, making it difficult to follow an execution trace.**
- **What should we use instead?**

Logging for Java: `Log4j`

- Enable logging at runtime without modifying the application binary:
  - Logging behavior can be controlled by editing a configuration file.
- A logger *hierarchy* makes it possible to control which log statements get printed.
- Verbosity can be set to multiple levels:
  - DEBUG < INFO < WARN < ERROR < FATAL.
- The log statements can be sent to a terminal, file, stream, socket etc.
- More info available here: `http://logging.apache.org`

8.3 Log4j Example

Log4j Example Code

```
import  org . apache . log4j . Logger;

public  class  SomeClass  {

   static     Logger  logger  =  Logger . getLogger( "SomeClass" ) ;

   public    SomeClass(){
      logger .debug( "Constructor       called . " ) ;
   };


   public  void  doSomething()
   {
      logger . info ( "Doing  something . " ) ;

      try{
         throw new Exception( "Something bad happened here . " ) ; }
      catch (Exception e) {
         logger . error (e . toString ( ) ) ;
      };
   };
};
```

```
import  org . apache . log4j . Logger;

public  class  LogDemo  {

   static     Logger  rootLogger  =  Logger . getRootLogger ( ) ;

   public     static   void   main( String  args [ ] )          {
      rootLogger . info ( "Starting  demo  app" ) ;

      rootLogger .debug( "new SomeClass() " ) ;
      SomeClass o = new SomeClass ( ) ;

      rootLogger .debug( " calling SomeClass . doSomething() " ) ; o .
      doSomething ( ) ;
      rootLogger . info ( "Exiting  demo  app" ) ;
   };
};
```

Log4j: Low Verbosity

- **LogDemo.properties: verbosity set to *INFO* and *ERROR*.**

```
log4j . appender . stdout=org . apache . log4j . ConsoleAppender
log4j . appender . stdout . layout=org . apache . log4j . PatternLayout
log4j . appender . stdout . layout . ConversionPattern=%5p  [%t ]         (%F:%L)  − %m %n

log4j . rootLogger=INFO,   stdout
log4j . logger . SomeClass=ERROR
```

```
$  java  −Dlog4j . configuration=LogDemo. properties  LogDemo
    INFO  [main] (LogDemo. java:13)  − Starting  demo  app
 ERROR  [main] (SomeClass . java:22)  − java . lang . Exception :
         Something  bad  happened  here .
    INFO  [main] (LogDemo. java:21)  − Exiting  demo  app
```

Log4j: High Verbosity

- **LogDemo.properties: Set logging level to *DEBUG*.**

```
log4j . appender . stdout=org . apache . log4j . ConsoleAppender log4j .
appender . stdout . layout=org . apache . log4j . PatternLayout
log4j . appender . stdout . layout . ConversionPattern=%5p  [%t ]         (%F:%L)  − %m %n

log4j . rootLogger=DEBUG,   stdout
log4j . logger . SomeClass=DEBUG
```

```
$  java  −Dlog4j . configuration=LogDemo. properties  LogDemo
    INFO  [main] (LogDemo. java:13)  − Starting  demo  app
 DEBUG  [main] (LogDemo. java:15)  − new  SomeClass()
 DEBUG  [main] (SomeClass . java:11)  − Constructor  called .
 DEBUG  [main] (LogDemo. java:18)  − calling  SomeClass . doSomething()
    INFO  [main] (SomeClass . java:17)  − Doing  something .
 ERROR  [main] (SomeClass . java:22)  − java . lang . Exception :
         Something  bad  happened  here .
    INFO  [main] (LogDemo. java:21)  − Exiting  demo  app
```