A Crash Course in Java

Horstmann Chapter 1

Unit 1
CS 3354
Spring 2017

Jill Seaman

A simple java class

Greeter.java

```
public class Greeter
{
   public Greeter(String aName)
   {
      name = aName;
   }
   public String sayHello()
   {
      return "Hello, " + name + "!";
   }
   private String name;
}
```

A driver

GreeterTester.java

```
public class GreeterTester
{
   public static void main(String[] args)
   {
     Greeter worldGreeter = new Greeter("World");
     String greeting = worldGreeter.sayHello();
     System.out.println(greeting);
   }
}
```

Compilation

- To compile the program enter at the prompt (Unix or Dos) (Greeter.java and GreeterTest.java must be in the current directory):

```
javac GreeterTester.java
```

  - ✦javac is the java compiler
  - ✦Greeter.java is automatically compiled since GreeterTester requires it.
  - ✦If successful, this command creates the files Greeter.class and GreeterTester.class in the same directory
  - ✦ the *.class files contain platform-independent bytecode
  - ✦bytecode is interpreted (executed) by a Java Virtual Machine (JVM), and will run on a JVM installed on **any** platform
  - ✦The program does NOT need to be recompiled to run on another platform.

Execution

- To run the program enter at the prompt (Unix or Dos):

```
workspace jill$ java GreeterTester
Hello World!
workspace jill$
```

  - This runs the java bytecode on a Java Virtual Machine.
  - The java tool launches a Java application. It does this by starting a Java runtime environment, loading the specified class, and invoking that class's **main** method.
  - The main method must be declared public and static, it must not return any value, and it must accept a String array as a parameter.

Java Platform

- a bundle of related programs that allow for developing and running programs written in the Java programming language
- two distributions:
  - Java Runtime Environment (JRE) contains the part of the Java platform required to run Java programs (the JVM)
  - Java Development Kit (JDK) is for developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

Editions of Java

- Different editions of java target different application environments
  - Java Platform, Micro Edition (Java ME) — targeting environments with limited resources.
  - Java Platform, Standard Edition (Java SE) — targeting workstation environments.
  - Java Platform, Enterprise Edition (Java EE) — targeting large distributed enterprise or Internet environments.
- Each edition offers slightly different libraries (APIs) suited for the given environment.
- API: Application Programming Interface: the specification of the interface.

Releases of Java

- Different releases of Java
  - JDK 1.0 (1996) Codename: Oak
  - JDK 1.1 (1997)
  - J2SE 1.2 (1998)
  - J2SE 1.3 (2000)
  - J2SE 1.4 (2002)
  - J2SE 5.0 (2004) (1.5)
  - Java SE 6 (2006) (1.6)
  - Java SE 7 (2011) (1.7)
  - Java SE 8 (2014) (1.8) (I have this one)

Principles

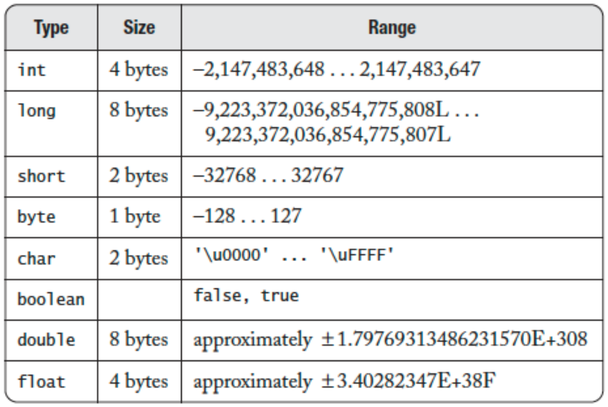
- There were five primary goals in the creation of the Java language:
  - ✦It should be "simple, object-oriented and familiar"
  - ✦It should be "robust and secure"
  - ✦It should be "architecture-neutral and portable"
  - ✦It should execute with "high performance"
  - ✦It should be "interpreted, threaded, and dynamic"

Features

- Interesting features of Java
  - ✦Object-oriented: everything is an object
  - ✦Inheritance
  - ✦Polymorphism: can use a subclass object in place of the superclass
  - ✦Garbage collection (dynamic memory allocation)
  - ✦Exception handling: built-in error handling
  - ✦Concurrency: built-in multi-threading
  - ✦Persistence: support for saving objects' state between executions
  - ✦Platform independence: supports web programming

Primitive types

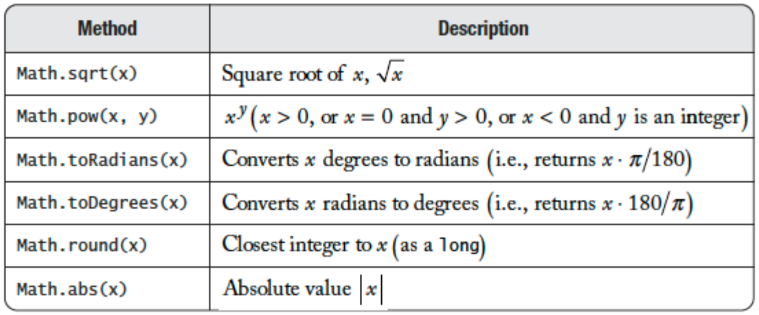
- These are NOT objects
- Size is not machine-dependent, always the same



Type	Size	Range
`int`	4 bytes	–2,147,483,648 . . . 2,147,483,647
`long`	8 bytes	–9,223,372,036,854,775,808L . . . 9,223,372,036,854,775,807L
`short`	2 bytes	–32768 . . . 32767
`byte`	1 byte	–128 . . . 127
`char`	2 bytes	`'\u0000' ... '\uFFFF'`
`boolean`		`false, true`
`double`	8 bytes	approximately ±1.79769313486231570E+308
`float`	4 bytes	approximately ±3.40282347E+38F

Math functions

- These functions are from the Math library class
- The parameters are numbers



Method	Description
`Math.sqrt(x)`	Square root of x, $\sqrt{x}$
`Math.pow(x, y)`	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)
`Math.toRadians(x)`	Converts x degrees to radians (i.e., returns $x \cdot \pi/180$)
`Math.toDegrees(x)`	Converts x radians to degrees (i.e., returns $x \cdot 180/\pi$)
`Math.round(x)`	Closest integer to x (as a `long`)
`Math.abs(x)`	Absolute value $\|x\|$

To call the sqrt function: `double y = Math.sqrt(x);`

Control flow in Java (same as C++)

- if-else

```
if(Boolean-expression)
  statement
else
  statement
```

```
if(Boolean-expression)
  statement
```

- while, do-while, and for

```
while(Boolean-expression)
  statement
```

```
do
  statement
while(Boolean-expression);
```

```
for(initialization; Boolean-expression; step)
  statement
```

- break and continue
- switch statement like C++
- statement can be multiple statements inside braces { }

Classes in Java, fields

- A Class defines a type with fields (data) and methods (operations)
- Fields can be objects or primitives

```
class ClassA {
  int i;
  Weeble w;
}
```

- Can create an object of this class using new:

```
ClassA a = new ClassA();
```

- Fields are accessible using dot operator

```
a.i = 11;
a.w = new Weeble();
```

Classes in Java, methods

- Methods in Java determine the messages an object can receive.
- They are functions that the object can execute on itself
- Syntax is very similar to C++:

```
class ClassA {
  int i;
  Weeble w;
  int mult (int j) {
     return i*j;
  }
}
```

- Methods are accessible using dot operator:

```
ClassA a = new ClassA();
a.i = 10;
int x = a.mult(4);
```

All objects in Java are really references

- Everything is treated as an object, using a single consistent syntax.
- However, the identifier you manipulate is actually a “reference” to an object (implemented as a pointer):

```
Greeter s;  //this is just a ref, a pointer
```

- Can assign `null` to object variables:

```
s = null;
```

- Dereferencing `null` causes a NullPointerException

```
s.setName(“Dave”);
```

- Note: references are on the run-time stack, objects are in heap.

Objects in Java versus objects in C++:

- Given this code in **Java**:

```
ClassA a = new ClassA();
a.i = 10;
int x = a.mult(4);
```

- This is the equivalent code in **C++**:

```
ClassA *a = new ClassA;
a->i = 10;
int x = a->mult(4);
```

- You cannot translate the following **C++** code to Java, because Java does not have statically allocated objects.

```
ClassA a;
a.i = 10;
int x = a.mult(4);
```

Assignment in Java

- Assignment in Java is like in C++
  - ✦For primitive types, values are copied

```
int a;
a = 10;
```

  - ✦For objects, the reference is copied so both variables refer to the same object.

```
Weeble b = new Weeble();
Weeble a;
a = b;    // a and b refer to same Weeble object
```

  - ✦changes to a will also affect b

Operators in Java

- Mathematical operators, same as C++

```
+ - * / %
++  --
+= -= *= /= %=
```

  - ✦integer division truncates, like C++
- Relational operators yield boolean result (not int)

```
< > <= >= == !=
```

  - ✦== over objects tests the value of the reference (the pointers)
- Logical operators

```
&&  ||  !
```

- String + is concatenation: this yields a new String object:

```
"abc" + "def"
```

```
"abcdef"
```

this

- The this keyword—which can be used only inside a method—produces a reference to the object the method has been called on.
  - ✦ in Java it's a reference (not a pointer)

```
class ClassA {
   int i;
   void seti(int i) {
      this.i = i;
   }
}
```

```
ClassA x = new ClassA();
x.seti(10);
//inside seti, "this" is equal to x
```

- It can also be used to call a constructor from another constructor (Unlike C++):

```
class ClassA {
   int i;
   ClassA(int i)
   {  this.i = i; }
   ClassA()
   {  this(0); } // calls ClassA(0)
}
```

Parameter Passing in Java

- Java uses call by value:
  - ✦For primitive types, values are copied to the function parameter
  - ✦For objects, the **address** of the object is copied to the function parameter
- Objects **can** be changed by calling mutators on the parameter

```
public class Greeter {
  String name;
  public void Greeter(String name) {
    this.name = name;
  }
  public void copyNameTo(Greeter other) {
    other.name = this.name;      //changes name of other
  }
}
```

```
Greeter worldGreeter = new Greeter("World");
Greeter dave = new Greeter("Dave");
worldGreeter.copyNameTo(dave);     //now both are “World”
```

Parameter Passing in Java

- a method can never update the contents of a variable that is passed as a parameter:

```
public class Greeter {
  public void copyLengthTo(int n)  {
    n = name.length();
  }
  public void copyGreeterTo(Greeter other) {
    other = new Greeter(name);
  }
. . .
}
```

```
int length = 0;
Greeter worldGreeter = new Greeter("World");
Greeter dave = new Greeter("Dave");
worldGreeter.copyLengthTo(length); //does not change length
worldGreeter.copyGreeterTo(dave);  //does not change dave
```

Packages

- Classes can be grouped into packages.
- Package names are dot-separated identifier sequences

```
java.util
javax.swing
com.sun.misc
```

- You place a class inside a package by adding a package statement at the beginning of the file:

```
package myPackage;
public class SmallBrain { … }
```

  - ✦Other classes (outside of myPackage) wanting access to SmallBrain must import myPackage, or fully specify it: `myPackage.SmallBrain`.

```
package anotherPackage;
import myPackage.*;
. . .
SmallBrain a;  // or myPackage.SmallBrain
```

Packages and Directories

- Package names must match subdirectory names and structure.
- To put your classes in a package called xx.myPackage:
  - ✦Declare the package on the first line of each java file

```
package xx.myPackage;
import ....
public class SmallBrain { ....
```

  - ✦Put all the files in package xx.myPackage in the following directory: ...src/xx/myPackage
  - ✦Make src the current directory: `cd ...src`
  - ✦To compile: `javac xx/myPackage/*.java`
  - ✦To run: `java xx.myPackage.ClassA`

Assuming ClassA contains a main method

Accessing classes from libraries

- In Java libraries, elements are grouped into packages
- Packages have dotted path names (like internet domains)
- To use a class from a package, import the qualified class name:

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

- Or import the entire package:

```
import java.util.*;
```

- You do not need to import classes from java.lang (like String or Math). These are imported automatically.

Java library documentation

- Online documentation for Java 1.8 API

  http://docs.oracle.com/javase/8/docs/api/

See the Gap 1 handout/exercise.

String

- The String class represents character strings.
- string literals like "abc" are implemented as instances of this class.
- strings are immutable (no methods to change their contents).
- Methods (many more available):
  - length() Returns the length of this string.
  - charAt(int i) Returns the `char` value at the specified index (but this cannot appear on the left of an assignment, you cannot change the string).
  - + for string concatenation (returns a new string)

```
String str = “abc”;
for (int i=0; i<str.length(); i++)
   System.out.println(str.charAt(i));
System.out.println(str+”def”);
```

String: substring and equals

- substring(i,e) computes a sub-piece of a string.
  - i is the position of the first character that you want to **include** in the substring and e is the first character that you **no longer want to include**.
  - "Hello".substring(1, 3) is the string "el"
- Since strings are objects, you need to use the **equals** method to compare whether two strings have the same contents.

```
String str = “el”;
String txt = “Hello”.substring(1,3);
if (str.equals(txt)) …  //OK, this is true
if (str==txt) … //NO this is false
```

  - The previous comparison fails because it compares the references. The references point to equivalent contents, but == compares the references. .equals compares the contents the references point to.

toString

- toString is a method that is defined by default for every class

```
public String toString();
```

- The String value returned should represent the data in the object.
- This makes it easy to output an object to the screen. The following are generally equivalent:

```
System.out.println(w);
```

```
System.out.println(w.toString());
```

- You can override the default definition by redefining toString for your class.

```
class ClassA {
   private int i;
   private double x;
   public String toString() {
      return (“i: “+i+” x: “+x);
   }
}
```

Wrapper classes

- Wrapper classes convert primitive type values to objects
  - ✦**Byte, Short, Integer, Float, Double, Boolean**, etc.
  - ✦In the java.lang package.
- Allows use of primitive values where Objects are required.
- Provides conversion functions between types.

```
int i = 50;
Integer mm = new Integer(i);
String k = mm.toString();  // k is now “50”
String k1 = Integer.toString(50); // the static toString method

int j = mm.intValue() + 5;  // j is now 55
int y = Integer.parseInt(k); // converts string to int
```

- compare() and compareTo(Integer) are defined as well.

Reading from the screen (Input)

- Scanner class (in java.util)
  - ✦Allows the user to read values of various types from a stream of characters.
  - ✦There are two constructors that are particularly useful: one takes an InputStream object as a parameter and the other takes a FileReader object as a parameter.

```
Scanner in = new Scanner(System.in);
// System.in is the InputStream associated with the keyboard

Scanner inFile = new Scanner(new FileReader(“myFile"));
// Creates a Scanner for a text file called myFile
```

Reading from the screen (Input)

- Useful Scanner methods:
  - ✦int nextInt() Returns the next token as an int. If the next token is not an integer, InputMismatchException is thrown.
  - ✦long nextLong() Similar
  - ✦float nextFloat() Similar
  - ✦double nextDouble() Similar
  - ✦String nextLine() Returns the rest of the current line, excluding any line separator at the end.
  - ✦boolean hasNextInt() Returns true if the next token in this scanner's input can be interpreted as an int value using the nextInt() method.
  - ✦hasNextLong(), hasNextFloat(), etc.

Reading from the screen (Input)

- Example using a Scanner with System.in:

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter the quantity: ");
int i = sc.nextInt();
System.out.println("Enter the price: ");
double price = sc.nextDouble();
System.out.println("Enter the name: ");
sc.nextLine();  //skip to end of previous line, after price
String name = sc.nextLine();
```

Writing to the screen (Output)

- System.out (in java.lang)
  - ✦System.out is a PrintStream, used to print characters.
  - ✦A PrintStream provides the ability to print **representations of various data values** conveniently.
- println(x) and print(x)
  - ✦Methods of PrintStream (see API website for details)
  - ✦Overloaded to print all the various data types.
  - ✦Often uses the default toString() method of the wrapper classes.
    - – for example, `Integer.toString(int i)` to print an int
  - ✦The difference between print( ) and println( ) is that the latter adds a newline when it’s done.

Writing to the screen: Formatting

- DecimalFormat class, used to format decimal numbers
  - ✦`DecimalFormat(String pattern)` Creates a DecimalFormat using the given pattern.
  - ✦format(x) produces a string by formatting an item (x) according to the objects pattern.
  - ✦The following characters have special meaning in a pattern (other characters are taken literally, appearing in the string unchanged).

`0`	digit (left-padded with zeros)
`#`	digit, zero shows as absent (no 0 padding)
`.`	decimal separator
`,`	Grouping separator
`E`	Separates mantissa and exponent in scientific notation
`%`	Multiply by 100, show as percent

Formatting example

```
import java.text.*;

class FormatOut {
     public static void main(String args[])  {
        int []  iArray = {1, 12, 123};
        float []  fArray = {1.1F, 10.12F, 100.123F};
        double []  dArray = {1.1, 10.12, 100.1234, 1000.1239};

        DecimalFormat dfi = new DecimalFormat("#00");
        DecimalFormat dff = new DecimalFormat("#00.00 float");
        DecimalFormat dfd = new DecimalFormat("#000.000");

        for (int i = 0; i < iArray.length; i++)
            System.out.println(dfi.format(iArray[i]));

        for (int i = 0; i < fArray.length; i++)
            System.out.println(dff.format(fArray[i]));

        for (int i = 0; i < dArray.length; i++)
            System.out.println(dfd.format(dArray[i]));
    }
}
```

Formatting example

- Output from running FormatOut:

```
01
12
123
01.10 float
10.12 float
100.12 float
001.100
010.120
100.123
1000.124
```

ArrayList class

- A Generic class: ArrayList<E> contains objects of type E
- Must specify the element types (base type) when declaring:

```
ArrayList<String> list = new ArrayList<String>();
```

  - ✦The base type must be a class (NOT primitive type).
- Basic methods:
  - ✦add(E x) Appends the specified element to the **end** of this list. Starts at position 0, increases size by 1.
  - ✦get(int i) Returns the element at the specified position in this list.
  - ✦set(int i, E x) changes element in position i to x.
  - ✦size() Returns the number of elements in this list (not the capacity).

ArrayList class

- ArrayList increase in size as needed automatically
- These methods insert and remove from the middle:
  - ✦add (int i, E x) inserts x at position i, after shifting all the elements from i to the end up by one location
  - ✦remove(int i) Removes the element at the specified position in this list, and closes the gap.
- ArrayList can be iterated over using a “for-each” loop:

```
ArrayList<String> list = new ArrayList<String>();
//Some code here to fill the list
for (String s : list)
   System.out.println(s); //does this for each String in list
```

  - ✦General syntax is: `for (BaseType var : arrayList) stmt`

Arrays in Java

- Arrays can store objects of any type, including primitives.
- Array length is fixed, array variable is a reference (an object)

```
int[] numbers = new int[10]; //all initialized to 0
```

- Arrays have bounds checking
  - ✦unable to access memory outside its block (using the array): runtime error
- Arrays are objects
  - ✦member **length** returns size of array
  - ✦can access elements using [x]

```
int[] c = { 0, 1, 4, 9, 16 };     //constructs+initializes
for(int i = 0; i < c.length; i++) //can also use foreach loop
  System.out.println(c[i]);
```

static keyword

- When a field or method is declared static, it means that data or method is not tied to any particular object instance of that class
- Instances of the class share the same static fields
- Static methods may not access non-static fields

```
class StaticFun {
  static i = 11;
  static void incr () { i++; }
}
```

- Static fields and methods may be accessed without instantiating any objects by using the class name, or from an existing object.

```
StaticFun.i = 100;
StaticFun sf = new StaticFun();
sf.incr();
```

The final keyword

- Java's final keyword has slightly different meanings depending on the context, but in general it says "This cannot be changed."
- Data
  - ✦To create named constants (primitive type):

```
public static final int VAL_THREE = 39;
```

  - ✦Use static so the class does not recreate it for each instance
  - ✦If you create an object that is final, it only means the reference cannot change, but the contents of the object itself could

```
private final Value v2 = new Value(22);
```

  - ✦Cannot assign v2 to something else, but you could change its fields

```
v2.setValue(25);
```

Javadoc

- javadoc: a tool to extract comments embedded in source code and put them in a useful form:
  - ✦HTML files, viewable from a browser.
  - ✦Can regenerate the HTML files whenever the comments/code change.
- Uses a special comment syntax to mark the documentation inside the source code
- javadoc also pulls out the class name or method name that adjoins the comment(s).
- html files are similar to the online Java API documentation.
- Purpose is to document the public **interface**: the class names and public methods.

Javadoc syntax

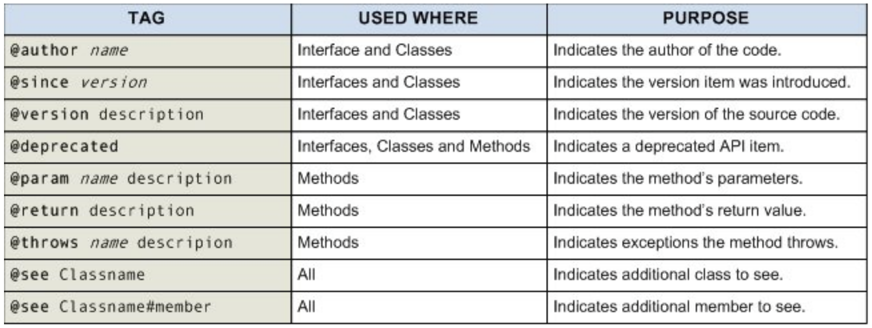
- The javadoc commands occur only within `/** … */` comments
  - ✦Note the initial double asterisks, normal comments have only one.
- Each javadoc comment must precede the class definition, instance variable definition or method definition that it is documenting.

```
/** A class comment */
public class DocTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

- The javadoc comments may contain the following:
  - ✦embedded html code, especially for lists and formatting code snippets
  - ✦"doc tags": special keywords that begin with @ that have special meaning to the javadoc tool.

Javadoc tags

- This table summarizes the more commonly used tags.



TAG	USED WHERE	PURPOSE
* @author *name*	Interface and Classes	Indicates the author of the code.
@since *version*	Interfaces and Classes	Indicates the version item was introduced.
@version description	Interfaces and Classes	Indicates the version of the source code.
@deprecated	Interfaces, Classes and Methods	Indicates a deprecated API item.
* @param *name* description	Methods	Indicates the method's parameters.
* @return description	Methods	Indicates the method's return value.
@throws *name* description	Methods	Indicates exceptions the method throws.
@see Classname	All	Indicates additional class to see.
@see Classname#member	All	Indicates additional member to see.

*required for this class
(use a separate @author tag for each author)

```
/**
 * A Container is an object that contains other objects.
 * @author Trevor Miller
 * @version 1.2
 * @since 0.3
 */
public abstract class Container {
  /**
   * Create an empty container.
   */
  protected Container() { }
  /**
   * Return the number of elements contained in this container.
   * @return The number of objects contained
   */
  public abstract int count();
  /**
   * Accept the given visitor to visit all objects contained.
   * @param visitor The visitor to accept
   */
  public abstract void accept(final Visitor visitor);
 /**
   * Determine whether this container is empty or not.
   * @return <CODE>true</CODE> if the container is empty:
   * <CODE>count == 0</CODE>, <CODE>false</CODE> otherwise
   */
  public boolean isEmpty() {
    return (this.count() == 0);
  }
}
```

Javadoc: generating the html files

- Use the javadoc command (from the JDK) to produce the html files:

```
javadoc -d api Container.java
```

- The -d option indicates a target directory for the html files
- Generates multiple .html files
- click on api/Container.html to see the result.

- For more details on javadoc, follow the javadoc links on the class website "readings" page.

Object serialization

- A process of transforming an object into a stream of bytes, to be saved in a file.
- Object serialization allows you to implement persistence:
- Persistence: when an object's lifetime is not determined by whether a program is executing; the object exists in between invocations of the program.
- The object's class must implement the `Serializable` interface.

```
public class Circle implements Serializable {  ...
```

✦If not, you get an exception: java.io.NotSerializableException: theClass

✦Note: there are no required methods to override

✦The field object types must be serializable too.

Object serialization: streams

- Java provides two object streams for serialization.
  - ✦These are both initialized given a FileOutputStream and a FileInputStream (respectively). The example shows how to initialize these given a file name.
- `ObjectOutputStream`
  - ✦The `writeObject()` method writes an object to the output stream, converting all the data in the object to bytes.
  - ✦All the field objects in the class must also be serializable
- `ObjectInputStream`
  - ✦The `readObject()` method reads an object from the input stream.
  - ✦The object was most likely written using `writeObject`
  - ✦You must cast the result to the correct object.

Serialization example: ZStudent.java

```
import java.io.*;

// Simple student class
class ZStudent implements Serializable {
    int no;
    String first, mid, last;  // Note these are serializable objects
    float ave;

    ZStudent() {};   // default constructor
    ZStudent(int no, String first, String mid, String last, float ave) {
        this.no = no;
        this.first = first;
        this.mid = mid;
        this.last = last;
        this.ave = ave;
    }

    public String display() {
        return (no + " " + first + " " + mid + " " + last + " " + ave);
    }
}
```

Serialization example: ObjFIO.java

```
import java.io.*;
import java.util.*;

class ObjFIO {
  public static void main(String[] args) {
    ArrayList<ZStudent> zstudents = new ArrayList<ZStudent>();
    zstudents.add(new ZStudent(50,  "Blue ", "M", "Monday   ", 50.0F));
    zstudents.add(new ZStudent(100, "Gray ", "G", "Tuesday  ", 60.0F));
    zstudents.add(new ZStudent(150, "Green", "G", "Wednesday", 70.0F));
    zstudents.add(new ZStudent(200, "Pink ", "P", "Thursday ", 80.0F));
    zstudents.add(new ZStudent(300, "Red  ", "R", "Friday   ", 90.0F));

    //the following code writes the objects to the file:
    try {
      FileOutputStream fos = new FileOutputStream("zStudentFile");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(zstudents); //ArrayList & contents are serializable
      fos.close();
    } catch (IOException e) {
      System.out.println("Problem with file output");
    }
```

Serialization example: ObjFIO.java cont.

```
    //the following code reads the objects from the file, then outputs
    try {
      FileInputStream fis = new FileInputStream("zStudentFile");
      ObjectInputStream ois = new ObjectInputStream(fis);

      ArrayList<ZStudent> students =
          (ArrayList<ZStudent>)ois.readObject(); // explicit cast reqd
      //demonstrate successful read:
      for (ZStudent zs : students)
          System.out.println(zs.display());

      fis.close();
    } catch (FileNotFoundException e) {
       System.out.println(“Cannot find datafile.”);
    } catch (IOException e) {
       System.out.println(“Problem with file input.”);
    } catch (ClassNotFoundException e) {
       System.out.println(“Class not found on input from file.”);
    }
  }
}
```

Serialization example

- Output from the example:

```
50 Blue  M Monday    50.0
100 Gray  G Tuesday   60.0
150 Green G Wednesday 70.0
200 Pink  P Thursday  80.0
300 Red   R Friday    90.0
```

- Why the try/catch syntax?
  - some of the library methods/constructors “throw exceptions” when they encounter a problem they can’t resolve.
  - If you call the method, you must catch the exceptions in catch blocks, and include code that indicates how you want the corresponding problem to be resolved.
  - we will discuss exception handling in more detail in the inheritance unit.