

Introduction to Java

(for C++ Programmers)

How to Think in Java

Everything is a class. Classes inherit from other classes. All classes inherit from the class Object. The Object class is a built-in Java class.

The First Java Program

Here is an example of a program that outputs "Hello World!"

Filename: First.java

```
public class First
{
    public static void main(String[] args)           // In C++ is: int main()
    {
        PrintClass printObject = new PrintClass("Hello world!"); // Must use new to create object
        printObject.display();
    }
}
```

Filename: PrintClass.java

```
public class PrintClass
{
    private String message;

    public PrintClass(String theMessage)           // Constructor
    {
        message = theMessage;
    }

    public void display()                          // Public method client can invoke
    {
        System.out.println(message);              // In C++ is: cout << message << endl;
    }
}
```

Notice that both the names of the files are the same name as the class – this is VERY important! The names are case sensitive too!

Creating Java Objects

To create an object you must use the new command, except for Strings – they're special and the Java compiler inserts a new in front for you. For example:

```
String s = new String("testing");
```

```
String s = "testing";           // Special case in Java. new is done automatically for you!
```

Both of the above examples create a String s, and give it a value.

Java Primitive types

There are primitive types in Java: int, double, boolean, char that act differently than Java objects. These primitive Java types are similar to the C++ types you have been working with.

Java Accessibility

Each class method/object/primitive has a description for its access: public, private, or protected.

public: Anyone can use it

private: only the class can use it

protected: only the class and classes that inherit from the class can use it

To use any method from a class you first must create an object of the class, then use the object with the method like we do in C++. For example;

```
Auto car = new Auto();      // C++ didn't require "new", Java does!  
car.fillUpTank();
```

The method fillUpTank cannot be used unless the object car exists.

```
Auto car;  
car.fillUpTank();          // this will produce an error! car doesn't refer to an object!
```

Using Reference variables in Java

All non-primitive variables in Java are references to objects; they do NOT contain the object's value, but a pointer to where the object is stored. In the example above the value of car is a pointer to where we can find the data store for an Auto object.

null can be used to make sure a reference variable does not point to anything. For example;

```
car = null;
```

car.fillUpTank() would cause an error because car does not reference (point to) any data.

The above following example

```
Auto car = new Auto();  
car = null;
```

would give C++ a memory leak, but in Java, the Java compiler does garbage collection and takes care of all the lost data via reference variables (C++ pointers). There is no delete command!

Static Methods

There are some methods you can use without creating an object. These are called static methods. In fact the starting point of any Java program

```
public static void main(String[] args)
```

is a static method.

You can use static methods without having an object. This is the normal way you wrote C++ functions. In Java this situation only arises when using a library of methods (e.g. Integer.parseInt(), Math.abs(), JOptionPane.showInputDialog()).

You call a static method by using the class name, not an object created from the class. For example, there is a Math class in Java that has many mathematical methods, including the square root method (sqrt). You would call this method

```
Math.sqrt(5);
```

Notice the naming protocol. You can tell sqrt is a static method because Math starts with a capital letter.

Java Naming Protocol

Nomenclature is important in Java in determining what is a class, what is an object, etc.

All class names start with an upper case letter

All variable names, and method names start with lower case letters

All final variables (C++ constants) use all upper case letters.

C++ Function is a Java Method

Notice that Java uses the word method instead of the word function!

All Java I/O is a String

Simple output:

```
int num = 7;
```

```
System.out.println("The value is " + num);
```

This will output

```
The value is 7
```

Notice that + is used to concatenate the string to the int.

Another important fact in Java: Everything is a String! In the above example the primitive variable num is converted into a String type by the Java compiler via the toString() method. Every class should have a toString() method to convert the object into a String. When outputting an object, its toString() is automatically called.

There is no just thing as simple input – but here goes:

```
import javax.swing.JOptionPane;
```

```
...
```

```
String input = JOptionPane.showInputDialog("Enter a number");
```

```
int num = Integer.parseInt(input);
```

This is called "Swing" input. A dialog box is displayed on the screen and you can enter your input. Remember all input is a String, therefore you must convert the String into an integer using the static parseInt method from the Integer class.

Wrapper Classes

Each primitive type has its own corresponding class, called a wrapper class.

int -> Integer

double -> Double

char -> Character

boolean -> Boolean

Each of these wrapper classes have methods to convert values back and forth between the primitive type and the class type. Wrapper classes are very important. Most Java classes cannot deal with primitive types, but all can handle class objects, therefore we need wrapper classes to deal with these common primitive types.

Type Casting

Most Java data structures work with the type Object. You will have to do a lot of *type casting* to change the Object to the type you want.

Comparing Objects

Comparing primitives in Java is just like comparing C++ types, you use the == for is equals to, and all the other corresponding inequality symbols.

Comparing objects is totally different. Each class should have an equals() method. For example to compare the value of two Strings

```
String s1 = "Mary";
String s2 = "Ott";
if (s1.equals(s2))
    ...
```

This above example works perfectly. The following example compares reference variables, not the values they point to, and does not perform the function you want.

```
if (s1 == s2)
    ...
```

Java Strings

There are many important String methods you should be aware of (e.g. equals, length, substring, indexOf, charAt). You can look these up in the Java API. The Java API should be opened at all times when programming in Java. The Java API contains all the Java classes and their methods, and a description on how to use them.

String Tokenizer

You should also be aware of how to use the StringTokenizer class. The class allows you to parse a String one token (i.e. word) at a time. For example:

```
import java.util.StringTokenizer;

public class First
{
    public static void main(String[] args)
    {
        String s = "Test a big dog, not a small cat";
        StringTokenizer token = new StringTokenizer(s);
        while (token.hasMoreTokens())
        {
            String t = token.nextToken();
            System.out.println(t);
        }
    }
}
```

Java Arrays

Arrays in Java are similar to C++ arrays, except that Java arrays are safe. You never will be allowed to access an array location that does not exist. You must use the new command to create an array. For example to create a ten element integer array;

```
int[] theArray = new int[10];
```

You can also create multi-dimensional arrays the same way.

ArrayList

ArrayList are similar to link list, but you do not have to create the nodes or links. You also can store any kind of object (object does NOT mean primitive type) in any location of the ArrayList. This means you can have a String in the first location and a Auto in the second location. The ArrayList is defined to be of type Object, the most general class in Java. Look in the Java API for the ArrayList methods (add, get, set, remove, size). I have an example using ArrayList below with Iterators.

Iterators

Iterators are another way to traverse a data structure. For example here is a program that uses the "old-fashion" way to traverse an ArrayList, and the technique using an iterator.

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

class TestIterators
{
    public static void main(String[] args)
    {
        ArrayList a = new ArrayList();
        a.add(new Integer(1));    // Convert primitive int to Integer object
        a.add(new Integer(2));
        a.add(new Integer(3));
        a.add(new Integer(4));

        // Using ArrayList methods to traverse the list
        for (int k = 0; k < a.size(); k++)
        {
            Integer entry = (Integer)a.get(k);
            System.out.println(entry);
        }

        // Using an iterator to traverse the list
        ListIterator it = a.listIterator();
        while (it.hasNext())
        {
            Integer x = (Integer)it.next(); // type cast to Integer object
            System.out.println(x);        // The toString() method is automatically called
        }
    }
}
```

Exceptions

An Exception is the way Java handles errors. For example:

```
try {
    int num = Integer.parseInt();
} catch (Exception(ex) {};
```

The above program statement `Integer.parseInt()` would produce an error if the `String` did not contain an integer value. The above code catches the error "thrown" by the `parseInt` method, but the above code doesn't do anything with the error returned (but it could have!) – see empty braces.

Parameters

In C++ plus there are two basic types of parameters: value and reference. In Java there is only one type of parameter: value. Now this can be deceiving. With primitive types a copy of the value is made, but with reference variables a copy of the reference is made, so really you have access to the object!

For example this method will **not** work as intended:

```
/**
 *      Swap the values passed
 */
public void swap(int x, int y) // x & y are primitive types
{
    int temp = x;
    x = y;
    y = temp;
}
```

The following example doesn't work either:

```
/**
 *      Swap the values passed
 */
public void swap(Auto x, Auto y) // x & y are reference variables
{
    Auto temp = x; // This swap only the local references
    x = y;
    y = temp;
}
```

Here is an example of how to use a reference:

```
/**
 *      Change the value passed
 */
public Auto swap(Auto x) // x & y are reference variables
{
    Auto car = new Auto();
    return car;
}
```

The above method will work if it is called this way:

```
Auto thecar = new Auto();
thecar = Change(theCar);
```

The main method has an interesting parameter. It will accept command line attributes: For example if the command to run the java program from the DOS prompt is entered:

```
DOS>java Test Mary 28
```

Mary and 28 will be passed to the program and stored in the String array parameter (e.g.)

```
public static void main(String[] args)
// args[0] contains Mary, and args[1] contains 28
```

Inheritance

Inheritance allows to add generally functionality from a class to a more specific class. For example all animals are mammals. Therefore the Animal class can inherit from the Mammal class, take all it's states and behaviors then add upon them. Java uses the extends command for inheritance. For example:

Filename: Person.java

```
public class Person
{
    private String name;

    public Person()
    {
        name = "";
    }

    public Person(String aName)
    {
        name = aName;
    }

    public String getName()
    {
        return name;
    }
}
```

Filename: Student.java

```
public class Student extends Person
{
    private int studentID;

    public Student()
    {
        super();           // Calls the Person class constructure
        studentID = 0;
        // or the two above lines can be replaced with the following line
        // this("", 0);
    }

    public Student(String aName, int anId)
    {
        super(aName);
        studentID = anId;
    }

    public int getId()
    {
        return studentID;
    }
}
```

Filename: TestInheritance.java

```
class TestInheritance
{
    public static void main(String[] args)
    {
        Student s = new Student("Mary Ott", 28);

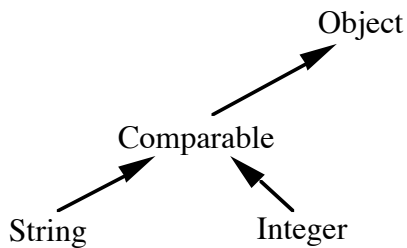
        System.out.println("Name is " + s.getName());    // Calls method from Person class
        System.out.println("Student ID is " + s.getId()); // Calls method from Student class
    }
}
```

Output is:

```
Name is Mary Ott
Student ID is 28
```

Hierarchy in Java

The chart listed below shows how four Java classes are related by inheritance:



From the above diagram we can see all String objects are Comparable objects (Strings are inherited from Comparables), but not all Comparables objects are String objects. Therefore it is a free cast upwards (e.g. String to Comparable), but you must explicitly cast downward (e.g. Comparable to String). For example;

```
Integer i;
Comparable c;

c = i;           // This is fine
i = c;           // This is an error
i = (Integer) c; // this is fine
```

It is important to know how all Java classes are related. For example from the above example, we now know that any method written in the Comparable class can also be used in the String & Integer class.

Important Methods

Every class should override the following methods:

- toString() – Used in the System.out.print(x), which translates to System.out.print(x.toString());
- equals(object) – Compare like objects for equality
- compareTo(objec) – Compare like objects for relativity
- hashCode() – Returns a unique hash value for the object
- iterator() – Returns a iterator object to traverse this object

Thinking about Casting and Comparable

```
public static void main(String[] args)
{
    Object o = null;
    Integer i = new Integer(3);
    String s = "hello";
    Comparable c = null;

    o = s;
}
```

This code compiles and runs without error. If the line below is added after `o = s` the compiler generates the error message shown.

```
o = s;
s = o;    // generates error message
```

Here are the error messages that result.

```
Casting.java:11: incompatible types
found   : java.lang.Object
required: java.lang.String
    s = o;
        ^
1 error
```

```
Compilation exited abnormally with code 1 at Wed Jun 26 10:30:31
```

This compilation error can be removed by casting as follows:

```
o = s;
s = (String) o;
```

Now consider the following changes to these two lines. The resulting code compiles successfully.

```
o = i;
s = (String) o;
```

However, although the program compiles, it generates the following runtime error message when executed.

```
prompt> java Casting
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
    at Casting.main(Casting.java:11)
```

Questions

Why is the cast in the line below necessary for the program to compile?

```
s = (String) o
```

Why do the following lines generate the runtime error shown in the explanation given above?

```
o = i;  
s = (String) o;
```

Both Integer and String implement the Comparable interface. Which one of the two lines below fails to compile, why, and how can it be fixed so that the code compiles and executes without error?

```
c = i;  
i = c;
```

The code below compiles and executes. What does it print and why?

```
Integer i = new Integer(3);  
String s = "hello";  
Comparable c = null;  
  
c = i;  
  
if (c.compareTo(i) < 0) {  
    System.out.println("less");  
}  
else {  
    System.out.println("not less");  
}
```

If the `if` statement test is changed as follows, the code compiles but generates a runtime error with a `ClassCastException`, why?

```
if (c.compareTo(s) < 0) {
    System.out.println("less");
}
else {
    System.out.println("not less");
}
```

If the `if` statement is changed as follows, the code compiles but generates a runtime error with a `NumberFormatException` for the string "hello", why?

```
if (c.compareTo(new Integer(s)) < 0) {
    System.out.println("less");
}
else {
    System.out.println("not less");
}
```

Suppose that `String s = "4";` is used instead of assigning "hello" as the value of `s`. What happens when the code in the previous problem is executed?

Explain why the code below compiles and executes without error (recall that `args` is the parameter to `main`).

```
Object[] list = {  
    new Integer(3), new Integer(4)  
};  
list = args;
```

Consider the code below.

```
Object[] list = {  
    new Integer(3), new Integer(4)  
};  
System.out.println(list[0].getClass().getName());  
list = args;  
System.out.println(list[0].getClass().getName());
```

When this program is run with a command-line argument the output is as shown below.

```
java Casting 1 2 3  
java.lang.Integer  
java.lang.String
```

Explain why, and explain the behavior of the run shown below in which no command-line arguments are supplied to the program.

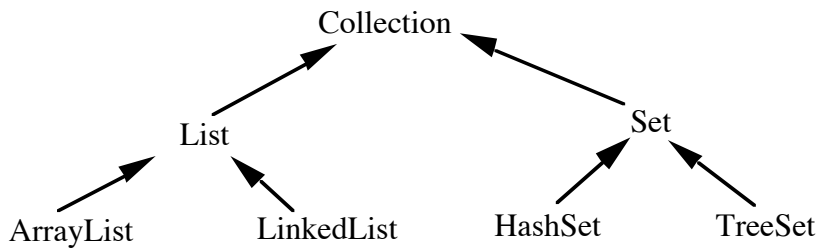
```
java Casting  
java.lang.Integer  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException  
    at Casting.main(Casting.java:29)
```

Explain why the code below compiles and executes, but why the cast in the second line is necessary.

```
Comparable[] clist = args;  
args = (String[]) clist;
```

Collections

Java supports numerous data types. Their inheritance hierarchy is listed below.



Sorting

The class Selection sort provides code to sort an array of Strings using the selection sort algorithm. The static method `sort` is reproduced below. Several questions follow that will focus discussion on the `Comparable` interface, the difference between arrays and `ArrayList` objects, and other features of Java.

```
public static void sort(String[] list)  
{  
    for(int j=0; j < list.length; j++) {  
        int min = j;  
        for(int k=j+1; k < list.length; k++) {  
            if (list[k].compareTo(list[min]) < 0) {  
                min = k;  
            }  
        }  
        String temp = list[min];  
        list[min] = list[j];  
        list[j] = temp;  
    }  
}
```

Questions

If the type of formal parameter `list` is changed from `String[] list` to `Comparable[] list`, what other changes in the method are needed to make the function compile?

The code below compiles and executes if formal parameter `list` is either a `String[]` or a `Comparable[]`, why?

```
String[] list = { "zebra", "apple", "bear", "monkey" };  
SelectionSort.sort(list);
```

Rewrite the function so that it sorts an `ArrayList` of `Comparable` objects. You'll need to use methods `ArrayList.get(int index)` to get the value stored at the specified index and `ArrayList.set(int index, Object o)` that stores `o` at the specified index.

```
/**  
 * Sorts list of Comparable objects using selection sort.  
 */  
  
public static void sort(ArrayList list)  
{
```