

Java and C++

- Both are “object oriented”
 - Class-based
- Java is interpreted, and garbage collected
 - C++ has more potential for memory leaks (you must explicitly delete objects)
- Java intended to be explicitly platform independent
- Similarities in syntax
- See Gary Shute’s document on Java for C Programmers (available from course calendar)

Outline

- Data types, Objects
- Compiling & running programs
- Control structures
- System objects & methods
- Console output & input
- Strings
- Exception handling
- Classes

Outline-2

- StringTokenizer
- File & directory naming & operations
- File I/O

Java

Type	Contains	Size, Coding, or Values
boolean	Truth value	true, false
char	Character	Unicode characters
byte	Signed integer	8 bit 2's compliment
short	Signed integer	16 bit 2's compliment
int	Signed integer	32 bit 2's compliment
long	Signed integer	64 bit 2's compliment
float	Real number	32 bit IEEE 754 floating point
double	Real number	64 bit IEEE 754 floating point

Java Objects

- Similar to C++ objects
 - Instance of a class
- Simple type variables (e.g., int)
 - Contain value copies
- Object variables
 - Contain references to objects
 - Two object variables can refer to the same object
- No explicit pointer types in Java

Compiling & Running

- At a UNIX system command line prompt (e.g., bulldog):
`javac class-name.java`
- Java compiler produces byte code
 - Not executable machine language
 - Needs to be interpreted
- You must have a class in the program file with the same name as the file (“class-name”)
- Running the program (at UNIX prompt):
`java class-name`
- CLASSPATH variable (best to put in .cshrc)

Hello World Program

```
// put this into a file named "A.java"  
public class A {  
    // need the following method in a 'main' program  
    public static void main(String args[]) {  
        //program code  
        System.out.println("Hello world");  
    }  
}
```

Control Structures: If-else

```
if (1 == 2) {  
    System.out.println("Oh my god!");  
} else {  
    System.out.println("Life is good.");  
}
```

While Loop

```
while (x > 0) {  
    System.out.print("x= " + x + "\n");  
    x--;  
}
```

*for loops, break & continue statement, do while,
switch*

Same as in C++ language

java.lang.System

- Provide platform-independent access to system functions
 - Class may not be instantiated
 - No “import” statement required

public static PrintStream err;

public static PrintStream out;

public static InputStream in;

- System standard input, output & error streams

public static void exit(int status);

- Exit the program

Console Output:

java.io.PrintStream

- Enables output of textual representations of Java data types
 - Arguments to methods take specific number of parameters
- **print** methods enable printing
 - Objects (with toString method)
 - String, char array, char, int, long float, double, boolean
- **println** with no parameters
 - Outputs newline
- **println** methods enable printing
 - Outputs newline after item
 - Objects (with toString method)
 - String, char array, char, int, long, float, double, boolean

Example

```
public class print {  
    public static void main(String args[]) {  
        int r = 100;  
        System.out.print("value of r: ");  
        System.out.println(r);  
        System.exit(0);  
    }  
}
```

Easier Technique: With Strings

```
public class print {  
    public static void main(String args[]) {  
        int r = 100;  
        System.out.println("value of r: " + r);  
        // overloaded "+": String concatenation  
    }  
}
```

Example 2

```
public class print {  
    public static void main(String args[]) {  
        int r = 100;  
        char x = 'a';  
        double d = 100.9;  
        System.out.println("value of r: "  
            + r + "\nx: " + x + "\nd: " + d);  
    }  
}
```

Console Input:

`java.io.BufferedReader`

- A bit complicated
- `java.io.BufferedReader`
 - Has a `readLine` method
 - Enables reading a `String` from the console
 - Reads strings up to, but not including, the newline
 - Reads spaces, tabs
- Constructor for `BufferedReader`
 - Requires a `Reader` object
 - Create an `InputStreamReader` object
 - Can pass the `InputStream System.in` to this constructor

```
// example of reading a string from the console
```

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

```
public class input {
```

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

```
        IOException
```

```
    {
```

```
        //program code
```

```
        System.out.print("Please enter a string: ");
```

```
        BufferedReader bufIn =
```

```
            new BufferedReader(new InputStreamReader(System.in));
```

```
        String s = bufIn.readLine();
```

```
        System.out.println("You entered: " + s);
```

```
    }
```

```
}
```

Strings: java.lang.String

- Sequences of characters
- *String* Objects are created by Java compiler when it encounters a sequence of characters in double quotes
- New strings can be created, but once created, they cannot be modified
 - use `StringBuffer` to modify contents of a string
- Methods
 - public static `String` `valueOf`
 - Convert basic types (e.g., `int`, `char`) to strings
 - Other methods include
 - `charAt`, `compareTo`, `equals`, `equalsIgnoreCase`

Example: *int* to *String* conversion

```
public class string {  
    public static void main(String args[]) {  
        int x = 100;  
        // Can't invoke a method on basic types  
        // System.out.println(x.toString());  
        System.out.println(String.valueOf(x));  
        // also, Integer.toString(x)  
    }  
}
```

String Equality Testing

- Use **equals** method
 - In general, for Object's, use methods for comparisons, and don't use relational operators (==, <, > etc)

```
String a = new String("abc");
```

```
String b = new String("abc");
```

```
if (a.equals(b)) {  
    System.out.println("Variables are equal.");  
} else {  
    System.out.println("Variables are NOT equal.");  
}
```

```
// Be careful with “==“ test
```

Recall, Testing Pointers in C Language

- When you use “==” operator with pointers
 - pointers are compared and not data referred to by the pointers

- Example

```
int *a;
int *b;
int c = 10;
a = &c;
b = &c;
if (a == b) {
    printf(“a and b are pointing to same memory object\n”);
}
```

- Tests if a and b are pointing at the same memory object, not if the values of the memory object they refer to are the same

In Java, References and the “==” (Equality) operator

- When you test Object's with the “==” equality operator
 - You are testing if they refer to the same Object
 - Not if the contents of the Object's are the same

String 'Equality' Testing with “==”

```
String a = new String("abc");  
String b = new String("abc");
```

```
if (a == b) {  
    System.out.println("Variables are equal.");  
} else {  
    System.out.println("Variables are NOT equal.");  
}
```

```
// Output: Variables are NOT equal.
```

Possibly Different Results with String Constants

```
String a = new String("abc");  
String b = new String("abc");  
  
if ("abc" == "abc") {  
    System.out.println("Strings are equal.");  
} else {  
    System.out.println("Strings are NOT equal.");  
}  
  
// Output: Strings are equal.
```

Why?

Explanation

- “abc” in each case is converted to a String object by compiler
- Compiler is smart, and in some cases will not create duplicate identical string constants
 - Saves a little memory
- Just uses reference for second instance of constant

Converting from Strings to Other Values

```
public class convert {  
    public static void main(String args[]) {  
        int x;  
        String s = "100";  
        // convert String to integer  
        x = Integer.parseInt(s);  
        x += 150;  
        System.out.println("x= " + x);  
    }  
}
```

More Conversions

```
String s = "100.1";
```

```
String m = "x";
```

```
// convert from String to a float value
```

```
float f = Float.parseFloat(s);
```

```
// convert from String to double value
```

```
double d = Double.parseDouble(s);
```

```
// convert from String to a character (char) value
```

```
char c = m.charAt(0);
```

Exception Handling

- Built into the Java language
- Exceptions
 - Signals indicating an unusual condition (e.g., error) has occurred
 - When exception occurs, we say the exception is *thrown*
 - Example
 - the *readLine* method of `java.io.BufferedReader` throws `IOException`
- Handling exceptions
 - Exceptions propagate up the block structure, and up the series of method calls
 - Exceptions must be either:
 - caught, or
 - declared in a *throws* clause of the calling method
 - Catching exceptions: try/catch/finally structure

// example of a throws clause

// here we are not catching the thrown exception, we are

// declaring it in the throws clause of the calling method

import java.io.*;

public class input {

 public static void main(String args[]) throws
 IOException

{

 //program code

 System.out.print("Please enter a string: ");

 BufferedReader bufIn =

 new BufferedReader(new InputStreamReader(System.in));

 String s = bufIn.readLine();

 System.out.println("You entered: " + s);

}

}

```
// example of catching an exception using a try/catch/finally block

import java.io.*;

public class exceptionExample {
    public static void main(String args[])
    {
        //program code
        BufferedReader bufIn =
            new BufferedReader(new InputStreamReader(System.in));
        String s = null;

        System.out.print("Please enter a string: ");
        try {
            s = bufIn.readLine();
        }
        catch (IOException e1) {
            System.out.println(
                "Error when reading line from console");
            // Could put some other statements here
        }

        System.out.println("You entered: " + s);
    }
}
```

Writing a Class

- A .java file can have one or more classes
 - Apparently, additional classes in the same file must be named without the “public” keyword
- Each class is self-contained
 - But may have abstract methods
 - May have subclasses (derived classes)
 - May use other classes
- Classes have methods and variables and constants
- Constructors are named with same name as class
- **finalize** methods can be used, per class
 - E.g., return resources to operating system
 - Automatically called by compiler when an object is destroyed, before garbage collecting the memory

Packages

- package statement
 - Can appear as the first statement of a source file
 - Specifies the package that the code is part of
 - Class must be at right place in hierarchy for it to be accessed by interpreter

Method and Variable Visibility Modifiers

- (default), public, private, protected
 - Default: a method or variable is public within same package (private elsewhere)

```
public class mult {  
    public static void main(String args[]) {  
        MyType y = new MyType();  
        y.blah = 10; // public  
        y.blah2 = 20; // public  
        y.blah3 = 50; // syntax error  
    }  
}
```

```
class MyType {  
    int blah;  
    public int blah2;  
    private int blah3;  
  
    MyType() {  
    }  
}
```

Modifiers: Visibility of Methods and Variables

- **public**
 - Visible to all methods
- **private**
 - Visible to methods within class
- **protected**
 - Visible to methods within class, package, subclasses in same package

Static vs. Instance Variables

- Variables or methods can be static
- Static variables are called *class variables*
- Each class has one variable
 - No separate class variable per instance

```
public class Static {  
    public static void main(String args[])  
    {  
        X blah = new X();  
        blah.foo++;  
        System.out.println("blah.foo= " + blah.foo); // 11  
        X blah2 = new X();  
        System.out.println("blah2.foo= " + blah2.foo); // 11  
    }  
}
```

```
class X {  
    static int foo = 10;  
}
```

```
public class TestStatic {
    public static void main(String args[])
    {
        X blah = new X(); X blah2 = new X();
        blah = null; blah2 = null;
        System.gc(); // suggest that garbage collector be called
    }
}

class X {
    // keep track of number of active instances
    static int numInstance = 0;

    X() { // constructor
        numInstance++;
    }

    protected void finalize() {
        numInstance--;
        System.out.println("number of instances: " + numInstance);
    }
}
```

Static vs. Instance Methods

- Instance methods
 - Invoked on instances of objects
 - E.g.,
`database.sort();` // invoke the sort method of database object
- Static methods
 - Invoked by preceding method name with name of class

Static Method Example

```
public class Static2 {  
    public static void main(String args[])  
    {  
        X.print();  
    }  
}  
  
class X {  
    static void print() {  
        System.out.println("This is a static method");  
    }  
}
```

Inheritance

- Java does not support multiple inheritance
 - I.e., a subclass cannot be derived from multiple superclasses
 - A class can implement interfaces, however
- Keyword is: *extends*
 - // assuming we have a predefined Shape class
 - public class Circle extends Shape {
 - // define Circle class
 - // inherits methods and variables from Shape class
 - }

Inheritance Exercise

- Write the beginnings of a Shape class, in Java
 - Instance variables:
 - x, y– giving the coordinates of the shape object
 - Color (java.awt.Color)
 - Constructors
 - set coordinates
 - No parameters; just give default coords
- Then, subclass (extend) the Shape class by making a Circle class
 - Add instance variable:
 - radius

```
// can't start this class using "java" program
```

```
import java.awt.Color;
```

```
public class Shape {
```

```
    protected float x, y; // coords of object
```

```
    protected Color c; // protected: visible within class, package, and package subclasses
```

```
    public Shape() {
```

```
        c = null;
```

```
        x = 0.0f; // need to explicitly give type here!
```

```
        y = 0.0f; // by default a real constant is a double
```

```
        // compiler complains at possible loss of precision
```

```
    }
```

```
    public Shape(float x, float y) {
```

```
        c = null;
```

```
        this.x = x; // implicit this argument
```

```
        this.y = y;
```

```
    }
```

```
}
```

```
class Circle extends Shape {
```

```
    protected float radius;
```

```
}
```

Abstract Classes & Methods

- Classes can be declared as abstract
 - Instances of the objects cannot be created
- Abstract classes must have at least one abstract method
 - If an abstract method is defined, and the class was not defined as abstract, then it is automatically abstract
- Lets you create classes which form a basis for inheritance

Exercise

- Modify the previous example of Shape and Circle
- Shape class should be abstract
- Include an abstract Draw method in the Shape class
- Circle class will need to override the Draw method and provide an implementation
 - (e.g., an empty implementation)

```
import java.awt.Color;

public abstract class Shape2 {
    protected float x, y; // coords of object
    protected Color c;

    public Shape2() {
        c = null;
        x = 0.0f; y = 0.0f;
    }

    public Shape2(float x, float y) {
        c = null;
        this.x = x;
        this.y = y;
    }

    public abstract void Draw();
}

class Circle extends Shape2 {
    protected float radius;
    public void Draw() {
    }
}
```

Interfaces

- Really, class definitions where *all* the methods are abstract
 - No variable data, however
- Interfaces may also contain *public final static* data
 - Class constants

```
public interface Shape {  
    public final static double PI = 3.141592654;  
    // also: java.lang.Math.PI  
    public abstract double area();  
    public abstract double volume();  
    public abstract String getName();  
}
```

java.util.StringTokenizer

- Can use this to break strings up into parts
 - E.g., if you get a string from the user console and it's composed of multiple integers
 - Use the StringTokenizer to pull the separate numbers out, then convert to integers

```
public StringTokenizer(String str, String delim);  
public StringTokenizer(String str);  
    // default: whitespace delimiters  
public boolean hasMoreTokens();  
public String nextToken();
```

Example 1

```
import java.util.StringTokenizer;

public class Tokens {
    public static void main(String args[]) {
        String s = "123 456 789";
        StringTokenizer tok = new StringTokenizer(s);

        while (tok.hasMoreTokens()) {
            String intString = tok.nextToken();
            System.out.println("string= " + intString);
        }
    }
}
```

OUTPUT:

string= '123'

string= '456'

string= '789'

```
import java.util.StringTokenizer; // Example 2
import java.io.*;

public class Tokens2 {
    public static void main(String args[]) throws IOException {
        BufferedReader bufIn =
            new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Please enter three integer values: ");
        String s = bufIn.readLine();

        StringTokenizer tok = new StringTokenizer(s);

        int x = Integer.parseInt(tok.nextToken());
        int y = Integer.parseInt(tok.nextToken());
        int z = Integer.parseInt(tok.nextToken());

        System.out.println("Integer values: " + x + " " + y + " " + z);
    }
}
```

java.io.File

- File & directory naming & other operations

public File(String path) throws IOException;

public boolean exists();

public boolean isFile();

public boolean isDirectory();

public boolean canWrite();

public boolean canRead();

public boolean delete();

Example

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

```
public class FileEx {
```

```
    public static void main(String args[]) throws IOException {
```

```
        File myFile = new File("foobar");
```

```
        if (myFile.exists()) {
```

```
            if (myFile.isFile()) System.out.println("It is a file");
```

```
            if (myFile.isDirectory()) System.out.println("It is a directory");
```

```
            if (myFile.canWrite())
```

```
                System.out.println("Can write the file/directory");
```

```
            if (myFile.canRead()) System.out.println("Can read the file/directory");
```

```
            myFile.delete();
```

```
        }
```

```
    }
```

```
}
```

java.io.FileInputStream

- Read a stream of bytes from a file

```
public FileInputStream(String name)
```

```
    throws FileNotFoundException;
```

```
public FileInputStream(File file);
```

```
public int read(byte [] b, int off, int len)
```

```
    throws IOException;
```

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

```
public class FileIn {
```

```
    public static void main(String args[]) throws  
        FileNotFoundException, IOException {
```

```
        byte b[] = new byte[1];
```

```
        FileInputStream f = new FileInputStream("foobar");
```

```
        f.read(b, 0, 1);
```

```
        System.out.println("byte read= " + b[0]);
```

```
        char c = (char) b[0];
```

```
        System.out.println("char value= " + c);
```

```
        f.read(b, 0, 1);
```

```
        System.out.println("byte read= " + b[0]);
```

```
        c = (char) b[0];
```

```
        System.out.println("char value= " + c);
```

```
    }
```

```
}
```

file 'foobar'

contains 'hi there'

OUTPUT:

byte read= 104

char value= h

byte read= 105

char value= i

java.io.FileOutputStream

- Write a stream of bytes to a file

```
public FileOutputStream(String name) throws  
    FileNotFoundException;
```

```
public FileInputStream(File file);
```

```
public int write(byte [] b, int off, int len)  
    throws IOException;
```

```
import java.io.*;

public class FileOut {
    public static void main(String args[]) throws
        FileNotFoundException, IOException {
        byte b[] = new byte[1];
        FileOutputStream f = new
            FileOutputStream("foobar");

        b[0] = 'a'; // supposed to require an explicit cast
        f.write(b, 0, 1);
        b[0] = 'b';
        f.write(b, 0, 1);
    }
}
```