Java for C++ Programmers
Why Java?

• **Object-oriented** (even though not purely…)

• **Portable** - programs written in the Java language are platform independent

• **Simpler development** – clever compiler: strong and static typing, garbage collection…

• **Familiar** – took the best out of C++.
Java highlights

• Static typing
• Strong typing
• Encapsulation
• Reference semantics by default
• One common root object
• Single inheritance of implementation
• Multiple inheritance of interfaces
• Dynamic binding
JVM – Java Virtual Machine

• JVM is an interpreter that translates Java bytecode into real machine language instructions that are executed on the underlying, physical machine

• A Java program needs to be compiled down to bytecode only once; it can then run on any machine that has a JVM installed
Java Virtual Machine
Running Java Programs

// file HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

> javac HelloWorld.java

    The compilation phase: This command will produce the java bytecode file HelloWorld.class

> java HelloWorld

    The execution phase (on the JVM): This command will produce the output “Hello World!”
The main() method

- Like C and C++, Java applications must define a `main()` method in order to be run.
- In Java, the `main()` method must follow a strict naming convention.
  - `public static void main(String[] args)`
- The `main()` method is always a `member function` of a class
  - No global functions
Types

• There are two types of variables in Java, *primitive* types (int, long, float etc.) and *reference* types (objects)
• In an assignment statement, the *value* of a primitive typed variable is copied
• In an assignment statement, the *pointer* of a reference typed variable is copied
Primitive Types

The Java programming language guarantees the size, range, and behavior of its primitive types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true,false</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unicode character</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit signed integers</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed integers</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed integers</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed integers</td>
</tr>
<tr>
<td>float</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td>double</td>
<td>64-bit floating point</td>
</tr>
<tr>
<td>void</td>
<td>-</td>
</tr>
</tbody>
</table>

The default value for primitive typed variables is zero bit pattern.
Reference Types

• Reference types in Java are \textit{objects}:
  – Identity: location on \textit{heap}
  – State: Set of \textit{fields}
  – Behaviour: Set of \textit{methods}

• The default value of reference typed variables is \textit{null}
Arrays

- Java arrays are objects, so they are declared using the new operator
- The size of the array is fixed

```java
Animal[] arr;  // nothing yet ...
arr = new Animal[4];  // only array of pointers
for(int i=0 ; i < arr.length ; i++) {
    arr[i] = new Animal();
    // now we have a complete array
}
```

- The length of the array is available using the field `length`.
Multidimensional arrays

- Multidimensional array is an array of arrays
- Size of arrays may not be the same

```java
Animal[][] arr; // nothing yet ...
arr = new Animal[4][]; // array of array pointers
for (int i = 0; i < arr.length; i++) {
    arr[i] = new Animal[i+1];
    for (int j = 0; j < arr[i].length; j++) {
        arr[i][j] = new Animal();
    }
}
```
Strings

- All string literals in Java programs, such as "abc", are instances of `String` class
- Strings are immutable
  - their values cannot be changed after they are created
- Strings can be concatenated using operator `+`
- All objects can be converted to `String`
  - Using `toString()` method defined in `Object`
- The class `String` includes methods such as:
  - `charAt()` examines individual character
  - `compareTo()` compares strings
  - `indexOf()` Searches strings
  - `toLowerCase()` Creates a lowercase copy
Flow control

Just like C/C++:

```c
if(x==4) {
   // act1
   i--;
} else {
   // act2
} while(i!=0);

int j;
for(int i=0;i<=9;i++) {
   j+=i;
}
```

```c
int i=5;
do {
   // act1
   i--;
} while(i!=0);
```

```c
char c=IN.getChar();
switch(c) {
case 'a':
case 'b':
   // act1
   break;
default:
   // act2
}
```
Java 1.5 – new **for-each** loop

```java
int[] array = new int[10];
// calculate the sum of array elements
for (int curr : array) {
    sum += curr;
}
```
Classes in Java

• In a *Java program*, everything must be in a class.
  – There are no global functions or global data

• Classes have *fields* (data members) and *methods* (member functions)

• Fields and can be defined as one-per-object, or one-per-class (static)

• Methods can be associated with an object, or with a class (static)
  – Anyway, methods are defined by the class for all its instances

• Access modifiers (private, protected, public) are placed on each definition for each member (not blocks of declarations like C++)
Class Example

```java
class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
    }
    public Rectangle(Point p, int w, int h) {
        origin = p; width = w; height = h;
    }
    public void setWidth(int width) {
        this.width = width;
    }
}
```
Inheritance

• It is possible to inherit only from one class.
• All methods are virtual by default

```java
class Base {
    void foo() {
        System.out.println("Base");
    }
}

class Derived extends Base {
    void foo() {
        System.out.println("Derived");
    }
}

public class Test {
    public static void main(String[] args) {
        Base b = new Derived();
        b.foo(); // Derived.foo() will be activated
    }
}
```
Interfaces

• Defines a *protocol* of communication between two objects

• Contains *declarations* but no implementations
  – All methods are *implicitly* public and abstract
  – All fields are *implicitly* public, static and final (constants).

• An interface can *extend* any number of interfaces.

• Java’s compensation for removing multiple inheritance. A class can *implement* many interfaces.
Interfaces - Example

```java
interface ISinger {
    void sing(Song);
}

interface IDancer {
    void dance();
}

class Actor implements ISinger, IDancer {
    // overridden methods MUST be public
    // since they were declared public in super class
    public void sing() { ... }
    public void dance() { ... }
}
```
Abstract Classes

• *abstract method* means that the method does not have an implementation
  - `abstract void draw();`

• *abstract class* a class that has at least one abstract method
  - Must be declared `abstract`
  - An abstract class is not-complete. Some parts of it need to be defined by subclasses.
  - Can’t create an object of an incomplete class: some of its messages will not have a behavior
Final

- **final data member**
  Constant member

- **final method**
  The method can't be overridden.

- **final class**
  ‘Base’ is final, thus it can’t be extended

```java
final class Base {
    final int i=5;
    final void foo() { }
}

class Derived extends Base { // Error
    // another foo ...
    void foo() {
        }
    }
```
Static Data Members

• Same data is shared between all the instances (objects) of a Class.
• Assignment performed on the first access to the Class.

```java
class A {
    public static int x_ = 1;
}

A a = new A();
A b = new A();
System.out.println(b.x_);
a.x_ = 5;
System.out.println(b.x_);
A.x_ = 10;
System.out.println(b.x_);
```

Output:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Static Methods

- Static method can access only static members
- Static method can be called without an instance.

```java
class TeaPot {
    private static int numOfTP = 0;
    private Color myColor_; 
    public TeaPot(Color c) {
        myColor_ = c;
        numOfTP++;
    }

    public static int howManyTeaPots() {
        return numOfTP;
    }

    public static Color getColor() {
        return myColor_;  // error
    }
}
```
Java Program Organization

• Java program
  – One or more Java source files

• Source file
  – One or more class and/or interface declarations.
  – If a class/interface is public the source file must use the same (base) name
    • So, only one public class/interface per source file

• Packages
  – When a program is large, its classes can be organized hierarchically into packages
    • A collection of related classes and/or interfaces
    • Classes are placed in a directory with the package name
Using Packages

– **Use fully qualified name**
  - A qualified name of a class includes the class’ package
  - Good for one-shot uses: `p1.C1 myObj = new p1.C1();`

– **Use import statement**
  - at the beginning of the file, after the package statement
  - Import the package member class:
    ```java
    import p1.C1;
    ...
    C1 myObj = new C1();
    ```
  - Import the entire package (may lead to name ambiguity)
    ```java
    import p1.*;
    ```

– classes from package `java.lang` are automatically imported into every class

– To associate a class with a package, put `package p as` the first non-comment statement in a source file
Visibility of Classes

- A class can be declared:
  - **public**: `new` is allowed from All packages
  - Default: `new` is allowed only from the same package

```java
package P1;
public class C1 {
}
class C2 {
}

package P2;
class C3 {
}

package P3;
import P1.*;
import P2.*;

public class DO {
    void foo() {
        C1 c1 = new C1();
        C2 c2 = new C2();  // error
        C3 c3;
        // error
    }
}
```
Visibility of Members

• A definition in a class can be declared as:
  – *public*
    • Can be accessed from outside the package.
  – *protected*
    • Can be accessed from derived classes
  – *private*
    • Can be accessed only from the current class
  – *default ( if no access modifier is stated )*
    • Usually referred to as "Package access".
    • Can be called/modified/instantiated only from within the same package.
The *Object* Class

- Root of the class hierarchy
- Provides methods that are common to all objects
  - `boolean equals(Object o)`
  - `Object clone()`
  - `int hashCode()`
  - `String toString()`
  - ...
Testing Equality

• The equality operator == returns true if and only if both its operands have the same value.
  – Works fine for primitive types
  – Only compares the identity of objects:

```java
Integer i1 = new Integer("3");
Integer i2 = new Integer("3");
Integer i3 = i2;

i1 == i1; // Result is true
i1 == i2; // Result is False
i2 == i3; // Result is true
```
Object Equality

• To compare between two objects the boolean equals(Object o) method is used:
  – Default implementation compares using the equality operator.
  – Most Java API classes provide a specialized implementation.
  – Override this method to provide your own implementation.

```java
i1.equals(i1)   // Result is true
i1 == i2;       // Result is false
i1.equals(i2)   // Result is true
```
Example: Object Equality

```java
public class Name {
    String firstName;
    String lastName;
    ... 
    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return firstName.equals(n.firstName) &&
        lastName.equals(lastName);
    }
}

More on the subtleties of `equals()` later in the course...
```
Wrappers

• Java provides objects which wrap primitive types.

    Integer n = new Integer(4);
    int m = n.intValue(); // java 1.4
    int k=n; // java 1.5 – autoboxing

    int l = Integer.parseInt(“123”); // l is 123
    String s1 = n.toString(); // s1 is “4”
    String s2 = “” + n; // s2 is “4”

• There is a wrapper class in java.lang package for every primitive type
  - Byte, Short, Integer, Float, Long, Double, Character
Garbage Collection

• C++: \texttt{delete} operator releases allocated memory.
  – Not calling it means memory leaks

• Java: no \texttt{delete}
  – Objects are freed automatically by the \textit{garbage collector} when it is clear that the program cannot access them any longer.
  – Thus, there is no "dangling reference" problem.
  – Logical memory leaks may still occur if the program holds unnecessary objects.
Handling input/output

- Class System provides access to the native operating system's environment through static methods and fields.
- It has three fields:
  - The out field is the standard output stream
    - Default is the same console, can be changed
    - Example: System.out.print("Hello");
  - The err field is the standard error output stream.
    - Used to display error messages
  - The in field is the standard input stream.
    - use it to accept user keyboard input.
    - Example: char c = (char) System.in.read();
Scanner Class

- Scanner objects parse primitive types and strings using regular expressions.
- To use Scanner: `import java.util.Scanner;`
- To create a scanner object: `new Scanner(input_source)`
  - Input source can be keyboard (System.in), files, string variables, etc.

Operations
- `nextInt()`, `nextBoolean()` - Returns value of indicated type
- `next()` - Returns sequence of characters up to next whitespace
- `findInLine()` - Looks for a specified pattern
- `hasNext()` - Returns true if this scanner has a token in its input.
  - Can be used to detect EOF.
Scanner Example

int i;
double d;
String s1, s2;
Scanner sc = new Scanner(System.in);
System.out.print("Enter an integer: ");
i = sc.nextInt();
System.out.println();
System.out.print("Enter a floating point value: ");
d = sc.nextDouble();
System.out.println();
System.out.print("Enter a string: ");
s1 = sc.next();
System.out.println();
System.out.print("Enter a string terminated by a new line: ");
s2 = sc.nextLine();
System.out.println();
System.out.println("Here is what you entered: ");
System.out.println(i);
System.out.println(d);
System.out.println(s1);
System.out.println(s2);
Collections

• A collection (a *container* in C++) is an object that groups multiple elements into a single unit.

• Containers can contain only objects
  – Autoboxing can help!

• The Java Collections Framework provides:
  – *Interfaces*: abstract data types representing collections.
    • allow collections to be manipulated independently of the details of their representation.
    • reusable data structures.
  – *Algorithms*: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.
- **Vector** and **HashTable** are old collection classes
  - Not deprecated for backward compatibility reasons
  - Use **ArrayList** and **HashMap** instead.
Map Interfaces and Classes

Map

SortedMap

AbstractMap

HashMap

LinkedHashMap

TreeMap
Iterate Through Collections

• An object that implements the `Iterator` interface generates a series of elements, one at a time
  – Successive calls to the `next()` method return successive elements of the series.
  – The `hasNext()` method returns true if the iteration has more elements
  – The `remove()` method removes from the underlying collection the last element that was returned by `next()`.
Set Example

Set set = new HashSet();  // instantiate a concrete set
set.add(obj);  // insert an elements
int n = set.size();  // get size
if (set.contains(obj)) {...}  // check membership

// iterate through the set using iterator
Iterator iter = set.iterator();
while (iter.hasNext()) {
    Object e = iter.next();
    ...
}

// iterate through the set using enhanced for loop
for (Object e : set) {
    ...
}
Class Collections

- Provides *static* methods for manipulating collections
  - `binarySearch()` searches a sorted list
  - `copy()` copies list
  - `fill()` replaces all list elements with a specified value
  - `indexOfSubList()` – looks for a specified sublist within a source list
  - `max()` returns the maximum element of a collection
  - `sort()` sorts a list
Class Arrays

• Provides *static* methods for manipulating arrays
  - `binarySearch()` searches a sorted array
  - `equals()` compares arrays
  - `fill()` places values into an array
  - `sort()` sorts an array
Resources

Java Tutorial -
http://java.sun.com/docs/books/tutorial/

Java 6 API Spec -
http://java.sun.com/javase/6/docs/api/