CMSC 341

Inheritance and the Collection classes
Inheritance in Java

- Inheritance is implemented using the keyword extends.

  ```java
  public class Employee extends Person {
      //Class definition goes here - only the
      //implementation for the specialized behavior
  }
  ```

- A class may only inherit from only one superclass.

- If a class is not derived from a super class then it is derived from `java.lang.Object`. The following two class declarations are equivalent:

  ```java
  public class Person {...}
  public class Person extends Object {...}
  ```
Polymorphism

- If Employee is a class that extends Person, an Employee “is-a” Person and polymorphism can occur.

```java
Person [] p = new Person[2];
p[0] = new Employee();
p[1] = new Person();
```

Creates an array of Person references
Polymorphism (cont.)

- However, a Person is not necessarily an Employee. The following will generate a compile-time error.
  
  ```java
  Employee e = new Person();
  ```

- Like C++, polymorphism requires general class on left of assignment operator, and specialized class on right.

- Casting allows you to make such an assignment provided you are confident that it is ok.

  ```java
  public void convertToPerson(Object obj)
  {
    Person p = (Person) obj;
  }
  ```
Virtual Method Invocation

- In Java, virtual method invocation is automatic. At runtime, the JVM determines the actual type of object a reference points to. Then, the JVM selects the correct overridden method for it.

- Supposing the `Employee` class overrides the `toString` method inherited from the `Person` class, then the `toString` method of the derived class, `Employee`, is invoked even though the reference is a `Person` reference.

```
Person p = new Employee;
p.toString();
```

Invokes the `toString` method of the `Employee` class
What is inherited by the subclass?

- All fields are inherited. Giving fields in super classes protected access allows methods of subclasses to reference the fields.
- All methods are inherited except for constructors.
- Inherited methods may be overloaded or overridden.
Constructors and Inheritance

- The superclass constructors are always called by the constructors of the subclasses, either implicitly or explicitly.
- To explicitly call the superclass constructor, in the first line of the subclass constructor make a call to the super method passing the appropriate parameters for the desired constructor.
The super Reference

- All overridden methods in a subclass also contain a reference to their corresponding methods in the superclass named super.
- The following code contains the use of the super reference to call the super class constructor and to use the implementation of the toString method of the superclass.
- Notice it also contains several uses of the this reference.
Super Class Example

```java
public class Person
{
    protected String name;
    private int age;
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
    public Person(String name){
        this(name, 0); // Call to other constructor
    }
    public String toString(){ return name;}
    public int getAge(){ return age;}
    public void setAge(int age){ this.age = age;}
    public void setName(String name){
        this.name = name;
    }
}
```
Subclass Example

```java
public class Employee extends Person {
    private double salary;
    public Employee(String name, int age, double sal) {
        super(name, age);               // Call to superclass constructor
        salary = sal;
    }
    public Employee(String name, double salary) {
        this(name, 18, salary);         // Call to constructor above
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double sal) {
        salary = sal;
    }
    public String toString() {         // Call to superclass toString method
        return super.toString()
            + " has a salary of " + salary;
    }
}
```
Polymorphism in Action

```java
class Test {
    public static void main(String []args) {
        Person [] people = new Person[3];
        people[0] = new Person("Sam");
        people[1] = new Employee("Jane", 45345.63);
        for(Person someone:people) {
            System.out.println(someone);
        }
    }
}
```

*println* invokes the *toString* method of the object the reference is pointing to, as if it were a pointer in C++ and the *toString* method were virtual.

Output:

- Sam
- Jane has a salary of 45345.63
- null
Abstract Classes and Methods

- Java also has abstract classes and methods like C++. If a class has an abstract method, then it must be declared abstract.

```java
public abstract class Node{
    String name;
    public abstract void type();
    public String toString(){ return name; }
    public Node(String name){
        this.name = name;
    }
}
```
Subclass of Abstract Class

Subclass of an abstract class must provide implementation for ALL the abstract methods or it must be declared abstract as well.

```java
public class NumberNode extends Node{
    int number;
    public void print(){
        System.out.println("Number node");
    }
    public NumberNode(String name, int num){
        super(name);
        number = num;
    }
    public String toString(){
        return super.toString() + " " + number;
    }
}
```
More about Abstract Classes

- Like C++, abstract classes can not be instantiated.
  
  // OK because n is only a reference.
  Node n;
  
  // OK because NumberNode is concrete.
  Node n = new NumberNode("Penta", 5);
  
  // Not OK. Gives compile error.
  Node n = new Node("Name");
Multiple Inheritance in Java

- There are always cases where a class appears to have characteristics of more than one class. Consider the following hierarchy.

```
  Person
   \  /
  Student   Employee
     \     
       \   
        \ TA
```

TA has characteristics of both a Student and an Employee
Interfaces

- Java only allows a class to extend one super class. It does not allow multiple inheritance like C++. However, to cope with the need for multiple inheritance, it created interfaces.
- An interface is like class without the implementation. It contains only
  - public, static and final fields, and
  - public and abstract method headers (no body).
A public interface, like a public class, must be in a file of the same name.

The methods and fields are implicitly public and abstract by virtue of being declared in an interface.

```java
public interface Employable {
    void raiseSalary(double d);
    double getSalary();
}
```
Interfaces (cont.)

- Many classes may implement the same interface. The classes may be in completely different inheritance hierarchies.
- A class may implement several interfaces.

```java
public class TA extends Student implements Employable
{
    /* Now TA class must implement the getSalary and the raiseSalary methods here */
}
```
Inheritance Progression

Inheritance of Implementation

- Point
  - 3DPoint
  - ColorPoint
- AbstractShape
  - Square
  - Circle

Inheritance of Interface

- Person
  - Student
  - Employee
  - Employable
  - TA
  - Bear

Note: In UML (Unified Modeling Language)
- Solid line means extends a superclass.
- Dotted line means implements an interface.
The Collections Framework

- The Java Collections Framework implements a lot of the functionality of the C++ Standard Template Library.
- It is a collection of interfaces, abstract and concrete classes that provide generic implementation for many of the data structures you will be learning about in this course.
The Collections Framework (cont.)

- All of the collection classes contain elements of type Object. Since every object in Java “is-a” Object, then we can create a collection of heterogeneous objects.

- Before we begin examining Collections, let us look at some of the interfaces the framework uses.
The Arrays class

- The java.util.Arrays class is a utility class that contains several static methods to process arrays of primitive and reference data.
  - `binarySearch` – searches sorted array for a specific value
  - `equals` – compares two arrays to see if they contain the same elements in the same order
  - `fill` – fills an array with a specific value
  - `sort` – sorts an array or specific range in array in ascending order according to the natural ordering of elements
Natural Order

- The natural order of primitive data types is known. However, if you create an array of type Object, how does the sort method know how to sort the array?

- One way is to pass a Comparator along with the array.

- A Comparator is an object that implements the java.util.Comparator interface.
The Comparator Interface

- The `compare` method must behave like C’s `strcmp` function. Returns
  - a negative number if o1 precedes o2,
  - a zero if they are equal, and
  - a positive number if o2 precedes o1.

```java
public interface java.util.Comparator
{
    int compare(Object o1, Object o2);
}
```
The Comparable Interface

- The other way to define the natural ordering of objects is by having the class implement the `Comparable` interface. The `compareTo` method also behaves like the `strcmp` method in C.

```java
public interface java.lang.Comparable {
    int compareTo(Object o);
}
```
Comparable Example

import java.util.*;
public class Fraction implements Comparable
{
    private int n;
    private int d;
    public Fraction(int n, int d){ this.n = n; this.d = d; }
    public int compareTo(Object o)
    {
        Fraction f = (Fraction) o;
        double d1 = (double) n/d;
        double d2 = (double)f.n/f.d;
        if (d1 == d2)
            return 0;
        else if (d1 < d2)
            return -1;
        return 1;
    }
    public String toString() { return n + "/" + d; }
}
Sort Example

public class FractionTest
{
    public static void main(String []args)
    {
        Fraction [] array = {new Fraction(2,3),
                             new Fraction(4,5), new Fraction(1,6)};
        Arrays.sort(array);
        for(Fraction f : array)
            System.out.println(f);
    }
}
Collections

- The Collections framework provides two inheritance hierarchies for its containers.
  - **Collection**
    - Operations for lists and arrays
  - **Map**
    - Operations for hashes and associative arrays
    - We will not be covering the Map interface in this course, but for more information on this topic, see Sun’s Collections tutorial at [http://java.sun.com/docs/books/tutorial/collections/index.html](http://java.sun.com/docs/books/tutorial/collections/index.html)
The Collection Interface

- Some of the most common methods of this interface are:
  - `add` – adds a new element
  - `remove` – removes an element
  - `size` – returns the number of elements
  - `isEmpty` – returns whether collection is empty
  - `contains` – checks whether collection contains an element
  - `iterator` – returns an `Iterator` object to traverse the Collection
List and Set Interfaces

- The Collection interface has two sub-interfaces.
  - The Set interface allows no duplicates to be added to the Collection.
  - The List interface allows for an ordered collection. Elements are traversed in the order which they are added to the list. Additional methods include:
    - get – returns the element at a specified index
    - indexOf – returns the index of a specified element
    - listIterator – returns a ListIterator object that traverses the list in both directions
Interface Hierarchy

Prior to the Collections framework, Java used a Vector class and a Hashtable class. These classes have been incorporated into the new framework.
Collection Example

```java
import java.util.*;
public class CollectionExample
{
    public static void main(String args[])
    {
        Collection a = new LinkedList();
        a.add(new Integer(5));
        a.add(new Integer(10));
        a.add(new Integer(3));
        a.add(new Integer(5));
        printAll(a);
    }
    public static void printAll(Collection c)
    {
        Iterator i = c.iterator();
        while(i.hasNext())
            System.out.println(i.next());
    }
}
```

Substitute with `HashSet` and `TreeSet` to see varying behavior.
Thread-safety

- One of the major improvements from the old Vector and Hashtable classes to the Collections framework was the separation of thread-safety from the implementation. The newer Collection classes are not thread-safe, but they can be converted to be thread-safe by using the Collections.synchronizedList, Set or Map methods.

List list = Collections.synchronizedList(new ArrayList());
Generics

- Since JDK 1.5 (Java 5), the Collections framework has been parameterized.
- A class that is defined with a parameter for a type is called a generic or a parameterized class. In C++, there were referred to as template classes.
- If you compare the Collection interface in the API for 1.4.2 to the one in version 1.5.0, you will see the interface is now called Collection<E>.
Collection <E> Interface

- The E represents a type and allows the user to create a homogenous collection of objects.
- Using the parameterized collection or type, allows the user to retrieve objects from the collection without having to cast them.

Before:
List c = new ArrayList();
c.add(new Integer(34));
Integer i = (Integer) c.get(0);

After:
List<Integer> c = new ArrayList<Integer>();
c.add(new Integer(34));
Integer i = c.get(0);
Generic Cell Example

public class CellDemo
{
    public static void main (String[ ] args)
    {
        // define a cell for Integers
        Cell<Integer> intCell = new Cell<Integer>( new Integer(5) );

        // define a cell for Floats
        Cell<Float> floatCell = new Cell<Float>( new Float(6.7) );

        // compiler error if we remove a Float from Integer Cell
        Float t = (Float)intCell.getPrisoner();
        System.out.println(t);
    }
}

class Cell< T >
{
    private T prisoner;
    public Cell( T p)
    { prisoner = p; }
    public T getPrisoner(){return prisoner; }
}
Dont’s of Generic Programming

- Like C++, you CANNOT use a parameter in a constructor.

\[
\begin{align*}
T \ text{ obj} &= \text{new T}(); \\
T \ [\ ] \ array &= \text{new T}[5];
\end{align*}
\]

- Like C++, you CANNOT create an array of a generic type.

\[
\begin{align*}
\text{Collection <Integer> c[\ ]} &= \text{new Collection<Integer>[10];}
\end{align*}
\]
Do’s of Generic Programming

- The type parameter must always represent a reference data type.
- Class name in a parameterized class definition has a type parameter attached.
  
  ```java
  class Cell<T>
  ```
- The type parameter is not used in the header of the constructor.
  
  ```java
  public Cell()
  ```
- Angular brackets are not used if the type parameter is the type for a parameter of the constructor.
  
  ```java
  public Cell3(T prisoner);
  ```
- However, when a generic class is instantiated, the angular brackets are used
  
  ```java
  List<Integer> c = new ArrayList<Integer>();
  ```
Bounding the Type

- You will see in the API a type parameter defined as follows `<? extends E>`. This restricts the parameter to representing only data types that implement E, i.e. subclasses of E

```java
boolean addAll(Collection<? extends E> c)
```
Bounding Type Parameters

- The following restricts the possible types that can be plugged in for a type parameter T.

  ```java
  public class RClass<T extends Comparable>
  ```

- "extends Comparable" serves as a bound on the type parameter T.
- Any attempt to plug in a type for T which does not implement the Comparable interface results in a compiler error message.
More Bounding

- In the API, several collection classes contain `<? super T>` in the constructor. This bounds the parameter type to any class that is a supertype of T.

```java
interface Comparator<T>
{
    int compare(T fst, T snd);
}

TreeSet(Comparator<? super E> c)
```
Generic Sorting

```java
public class Sort {
    public static <T extends Comparable<T>> void bubbleSort(T[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            for (int j = 0; j < a.length - 1 - i; j++) {
                if (a[j+1].compareTo(a[j]) < 0) {
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
                }
            }
        }
    }
}
```
Generic Sorting (cont.)

- Given the following:
  
  ```java
class Animal implements Comparable<Animal> { ... }
class Dog extends Animal { ... } 
class Cat extends Animal { ... } 
```

- Now we should be able to sort dogs if contains the `compareTo` method which compares animals by weight.

- BUT... bubblesort only sorts objects of type T which extend T. Here the super class implements Comparable.

- New and improved sort on next page can handle sorting Dogs and Cats.
public class Sort
{
    public static <T extends Comparable<? super T>>
    void bubbleSort(T[] a)
    {
        for (int i = 0; i < a.length - 1; i++)
            for (int j = 0; j < a.length - 1 - i; j++)
                if (a[j+1].compareTo(a[j]) < 0)
                    {
                        T tmp = a[j];
                        a[j] = a[j+1];
                        a[j+1] = tmp;
                    }
    }
}