Reusing Classes in Java

- **Composition**
  - A new class is composed of object instances of existing classes.
  - Fields of one class contain objects from another.
  - Name class can be made up of three Strings (first, middle, last), Student class can contain a Name object and other Strings.

- **Inheritance**
  - Creates a new class as an extension to an existing class.
  - New class adds code to the existing class the without modifying it, resulting in two classes (the original class and the extension).
  - All classes inherit from Java standard class `java.lang.object`.

Simple Example of Composition

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    private WaterSource source;
    SprinklerSystem() {
        System.out.println("SprinklerSystem");
        valve1 = "v1";
        source = new WaterSource();
    }
}
Simple Example of Inheritance

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        System.out.println(x);
    }
}

Output: Cleanser dilute() apply() scrub()

General convention

• Fields are private
  ✦ Not even subclasses should access these directly
• Methods are public
  ✦ This is so other classes, including subclasses can access them.
• Overriding a method:
  ✦ Writing a new instance method in the subclass that has the same signature as the one in the superclass.
  ✦ Any instance of the subclass will use the method from the subclass
  ✦ Any instance of the superclass will use the method from the superclass
  ✦ The subclass can call the superclass method using "super.method()"

Simple Example of Inheritance

public class Detergent extends Cleanser {
    // Change (override) a method:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Call base-class version
    }
    // Add methods to the interface:
    public void foam() { append(" foam()"); }
    // Test the new class:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute(); x.apply(); x.scrub(); x.foam();
        System.out.println(x);
        Cleanser.main(args);
    }
}

Output: Detergent dilute() apply() scrub() foam()

Some things you can do in a subclass

• The inherited fields (from the superclass) can be used directly, just like any other fields (unless they are private).
• You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
• You can declare new fields in the subclass that are not in the superclass.
• The inherited methods (from the superclass) can be used directly (unless they are private).
• You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
• You can declare new methods in the subclass that are not in the superclass.
Initialization

- Java automatically inserts calls to the (default) superclass constructor at the beginning of the subclass constructor.

```java
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}
class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}
public class Cartoon extends Drawing {
    public Cartoon() {
        System.out.println("Cartoon constructor");
    }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

Output:

```
Art constructor
Drawing constructor
Cartoon constructor
```

So constructors are not inherited, they are called from the constructors of the subclass.

More about inheritance

- “Upcasting”
  - The type of an object is the class that the object is an instance of.
  - Java permits an object of a subclass type to be treated as an object of any superclass type.
  - This is an implicit type conversion called upcasting

Any method taking a Game as an argument can also take a BoardGame

- When to use composition, when to use inheritance
  - Usually, composition is what you want
  - Use inheritance if you want the interface (public members) of the re-used (superclass) object to be exposed in the new class
  - Use inheritance if you want your new class objects to be able to be passed to methods expecting the re-used class (if you need upcasting).

Access specifiers (reminder)

- keywords that control access to the definitions they modify
  - public: accessible to all other classes
  - private: accessible only from within the class in which it is defined
  - package (unspecified, default): accessible only to other classes in the same package
  - protected: accessible to all classes derived from (subclasses of) the class containing this definition, even if the class is in another package. Note: protected also provides package access.
Polymorphism

• Upcasting:
  ✦ Permitting an object of a subclass type to be treated as an object of any superclass type.

• Polymorphism:
  ✦ The ability of objects belonging to different types to respond to method calls of the same name, each one according to an appropriate type-specific behavior.
  ✦ It allows many types (derived from the same superclass) to be treated as if they were one type, and a single piece of code to work on all those different types equally, yet getting type-specific behavior for each one.

Example continued

```java
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute); //upcasting to Instrument
        tune(violin); //upcasting to Instrument
    }
}
```

What is output?

```
Wind.play() Middle C
Stringed.play() Middle C
```

Polymorphism:

in tune, i is an Instrument, but it calls the play method based on the specific type of the object it receives.

Example:

• Wind, Stringed and Percussion are Instruments

```java
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        tune(flute); // No upcasting necessary
tune(violin);
    }
}
```

Output:

```
Wind.play() Middle C
Stringed.play() Middle C
```
But we do have upcasting and polymorphism:

- We can get the same effect with just one tune method

```java
public class Music {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Percussion snaredrum = new Percussion();
        tune(flute); // upcasting
        tune(violin);
        tune(snaredrum);
    }
}
```

- What would the output be if we did not have polymorphism?
- Note: C++ requires “virtual” keyword (on play()) to get polymorphism.

Dynamic (run-time) binding

- Given the definition of tune, how does the compiler know which definition of the play method to call? Instrument? Wind? Stringed?

```
public static void tune(Instrument i) {
    i.play("Middle C");
}
```

- It will differ depending on the specific type of each argument passed to i.
- This cannot be determined at compile time.

- Binding: connecting the method call to a method definition.
  - Static binding: done at compile time (play binds to Instrument.play)
  - Dynamic binding: at run-time, the JVM determines the actual type of i and uses its play() definition. It can vary for each invocation of tune.
  - If the actual type of i does not define “play()”, the JVM looks for the nearest definition in its superclass hierarchy.

Extensibility

- Lets go back to the polymorphic tune method, AND add some more methods and instruments

```java
class Instrument {
    void play(String n) {
        System.out.println("Instrument.play() " + n);
    }
    String what() { return "Instrument"; }
    void adjust() {}
}
class Wind extends Instrument {
    void play(String n) {
        System.out.println("Wind.play() " + n);
    }
    String what() { return "Wind"; }
    void adjust() {}
}
class Percussion extends Instrument {
    void play(String n) {
        System.out.println("Percussion.play() " + n);
    }
    String what() { return "Percussion"; }
    void adjust() {}
}
class Woodwind {
    void adjust() {}
}
class Brass {
    void adjust() {}
}
```

Extensibility part 1
Extensibility part 2

```java
class Stringed extends Instrument {
    void play(String n) {
        System.out.println("Stringed.play() " + n);
    }
    String what() { return "Stringed"; }
    void adjust() {}
}
class Brass extends Wind {
    void play(String n) {
        System.out.println("Brass.play() " + n);
    }
    String what() { return "Brass"; }
}
class Woodwind extends Wind {
    void play(String n) {
        System.out.println("Woodwind.play() " + n);
    }
    String what() { return "Woodwind"; }
}
```

Extensibility part 3

```java
public class Music3 {
    public static void tune(Instrument i) {
        i.play("Middle C");
    }
    public static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
}
```

Output:

- Wind.play() Middle C
- Percussion.play() Middle C
- Stringed.play() Middle C
- Brass.play() Middle C
- Woodwind.play() Middle C

We extended our system by adding methods and new subclasses, but we did NOT need to change (or add to) the tune function.

Abstract methods and classes

- Purpose of the Instrument class is to create a common interface (public methods) for its subclasses
  - No intention of making direct instances of Instrument
- An abstract class is a class that cannot be instantiated, but it can be subclassed
- It may or may not include abstract methods.
- An abstract method is a method that is declared without a method body (without braces, and followed by a semicolon), like this:
  ```java
  abstract void f(int x);
  ```
- If a class contains an abstract method, it **must** be declared to be an abstract class.

Abstract methods and classes, example

- Any class that inherits from an abstract class must provide method definitions for all the abstract methods in the base class.
  - Unless the derived class is also declared to be abstract
- The Instrument class can be made abstract:
  - No longer need “dummy” definitions for abstract methods
  - Programmer and compiler understand how the class is to be used.

```java
abstract class Instrument {
    private int i; // Storage allocated in each subclass
    abstract void play(String n); // subclass must define
    String what() {
        return "Instrument";
    }
    abstract void adjust(); // subclass must define
}
```
Interfaces

- In the Java programming language, an interface is a form or template for a class: all of its methods must be abstract (no method bodies).
- Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.
- Interfaces cannot have constructors (there are no fields to initialize).
- An interface is a “pure” abstract class: no instance-specific items.
- An interface may contain fields, but these are implicitly static and final (named constants).

To create an interface, use the interface keyword instead of the class keyword.
❖ The methods (and fields) are automatically public
❖ Interfaces are not instantiated.
❖ Interfaces cannot have constructors.
❖ An interface can be used as a type (for variables, parameters, etc).
❖ Java permits an object instance of a class that implements an interface to be upcast to the interface type.

Interfaces, example

```java
interface Instrument {
    void play(String n); // Automatically public
    String what();     // and abstract
    void adjust();     
}
class Wind implements Instrument {
    public void play(String n) {
        System.out.println("Wind.play() " + n);
    }
    public String what() { return "Wind";
    }
    public void adjust() {}    
}
class Percussion implements Instrument {
    public void play(String n) {
        System.out.println("Percussion.play() " + n);
    }
    public String what() { return "Percussion";
    }
    public void adjust() {}    
}
class Stringed implements Instrument {
    public void play(String n) {
        System.out.println("Stringed.play() " + n);
    }
    public String what() { return "Stringed";
    }
    public void adjust() {}    
}
class Brass extends Wind {
    public void play(String n) {
        System.out.println("Brass.play() " + n);
    }
    public String what() { return "Brass";
    }
}
class Woodwind extends Wind {
    public void play(String n) {
        System.out.println("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind";
    }
}
class Stringed {
    public static void tune(Instrument i) { //unchanged
        i.play("Middle C");
    }
    public static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
}
```

Output:
```
Wind.play() Middle C
Percussion.play() Middle C
Stringed.play() Middle C
Brass.play() Middle C
Woodwind.play() Middle C
```

The rest of the code is the same as before.
“Multiple Inheritance”

- A Class may have **only one** immediate superclass
  - But it may have many ancestors in the hierarchy
- A Class my implement **any number of** interfaces.
  - This allows you to say an x is an A and a B and a C

### Interface Multiple Inheritance example

```java
interface CanFight {
  void fight();
}
interface CanSwim {
  void swim();
}
interface CanFly {
  void fly();
}
class ActionCharacter {
  public void fight() {System.out.println("fight");}
}
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
  public void swim() {System.out.println("swim");}
  public void fly() {System.out.println("fly");}
}
public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
}
```

### Extending an Interface

```java
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
}
```

- Suppose that later you want to add a third method to DoIt:

```java
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
  boolean didItWork(int i, double x, String s);
}
```

- If you make this change, all classes that implement the old DoIt interface will break because they don’t implement the interface

### Extending an Interface

- Solution: you could create a DoItPlus interface that extends DoIt.

```java
public interface DoItPlus extends DoIt {
  boolean didItWork(int i, double x, String s);
}
```

- Now users of your code can choose to continue to use the old interface (DoIt) or to “upgrade” to the new interface (DoItPlus).
Interface or Abstract class?

- **Interface**
  - ✦ Pro: can be implemented by any number of classes
  - ✦ Con: each class **must** have its own code for the methods, common method implementations must be duplicated in each class
- **Abstract Class**
  - ✦ Pro: subclasses do not have to repeat common method implementations, common code is in the abstract superclass
  - ✦ Con: Cannot be multiply inherited.

Implementing the Java Comparable Interface

- Assume you want to sort an array or ArrayList of custom objects (instances of some class you created).
- The following static methods are available in the Java API:
  ```java
  void Collections.sort(List<T> list)    // for ArrayLists
  void Arrays.sort(Object [] a)          // for static arrays
  ```
- All elements in the list/array must implement the java.lang.Comparable<T> interface:
  ```java
  int compareTo(T o);     //T is your custom class
  ```
  Compares this object with the specified object (o) for order. Returns a negative integer, zero, or a positive integer when this object is less than, equal to, or greater than (respectively) the specified object.

Sorting with Comparable, example

```java
import java.util.*;
public class Student implements Comparable<Student> {
    private String name;
    private String major;
    private int idNumber;
    private float gpa;
    public Student(String name, String major, int idNumber, float gpa) {
        this.name = name;  this.major = major;
        this.idNumber = idNumber; this.gpa = gpa;
    }
    public String toString() {
        return "Student: " + name + " " +major + " " + idNumber + " " + gpa;
    }
    public int compareTo(Student rhs) {
        return name.compareTo(rhs.name);
    }
}
```

**Output:**

```
Before: 
Student: Doe, J Math 1234 3.6
Student: Carr, M CS 1000 2.7
Student: Ames, D Business 2233 3.7
After: 
Student: Ames, D Business 2233 3.7
Student: Carr, M CS 1000 2.7
Student: Doe, J Math 1234 3.6
```
Sorting with Comparable, sort by gpa

- To sort by gpa, redefine compareTo as follows:

```java
public int compareTo(Student rhs) {
    float rhsGpa = rhs.gpa;
    if (gpa < rhsGpa) return -1;
    if (gpa == rhsGpa) return 0;
    return 1;
} // or return Math.round(gpa - rhsGpa);
// or return (new Float(gpa)).compareTo(rhsGpa);
```

Output:

Before:
Student: Doe, J Math 1234 3.6
Student: Carr, M CS 1000 2.7
Student: Ames, D Business 2233 3.7

After:
Student: Carr, M CS 1000 2.7
Student: Doe, J Math 1234 3.6
Student: Ames, D Business 2233 3.7

Note: compareTo is already defined in the primitive-type wrapper classes