Object-Oriented Programming (OOP) 2

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

- **Static Variables**
  - Variables that are associated with the **class** itself rather than individual object
  - Can be access through the class using

```
ClassName.variableName
```

    or through the object using

```
objectName.variableName
```

  - To access from class or from outside of an object, the data must be **public**
  - Used when variables are shared among objects
    - See `StaticDemo.java`

Variable vs Method

- When should we use a **variable** and when should we use a **method**?
  - Variables should be used to store the basic properties of an object
    - Can be changed through **mutator** methods but should not become **obsolete**
  - Methods should be used to calculate/determine values using variables
    - We do not want to waste time calculating something that is set
    - However, if a value may change over time, it should be calculated

Copying Objects

- Sometimes, for various reasons, we need to make a copy of an object
- In Java, there are two primary ways of doing this:
  - Using a **copy constructor** for the class
    - This method takes an argument of the same class type and makes a copy of the object
    - Example

```
String newString = new String(oldString);
```

  - Using the **clone** method for aclass
    - This allows an object to **make a copy of itself**
    - It is a bit more complicated to use
    - We will defer to this to CS 0445

Copying Objects

- When copying objects, we always need to be aware of exactly **WHAT** is being copied:
  - **Shallow Copy**: Assign each instance variable in the old object to the corresponding instance variable in the new object
    - If the instance variables are themselves references to objects, those objects will be shared
    - See `ex12b.java` and `Score.java`
  - **Deep Copy**:
    - Copy primitive type normally
    - For reference types, do not assign the reference; rather **follow the reference** and copy that object as well
      - This process could proceed through many levels

Copying Objects

- **Deep copies** tend to be more difficult to implement than **shallow copies**
  - Need to follow references and make copies
  - Example: linked list
- Neither shallow nor deep is necessarily correct or incorrect
- It depends on the needs for a given class
- The important thing is to be aware of who your copies are being made and the implications thereof

Returning References from Methods

- We know a method can return only a single value
  - We did multiple values before (using array)
  - But all values must have the same type
- Note that the value (returned by a method) can be a reference to an object which can contain an arbitrary amount of data
- We already discussed **composition** where an object can contain references to other objects within it
- Question: If a method is to return a reference to an object within another object, do we:
  - Return a reference to the actual object OR
  - Return a reference to a copy of the object

Returning References from Methods

- What access do we need?
  - Are we just looking at the object, or do we need to mutate it?
  - If we want to mutate it, do we want the mutation to be local or should it impact the encompassing object?
  - Text suggests returning copies, but again, it depends on the goals
    - What do we want to do with it?
    - What if we need to update the data?
      - A reference gives us access to do this easily
  - Alternative is:
    1. Return a copy
    2. Delete the original (no two objects trying to modify the same object)
    3. Update the copy
    4. Reinsert it back

Returning References from Methods

- Keep in mind that **returning references to the original is more dangerous** than returning copies
  - If we accidentally modify the object via the returned reference, that will impact the original encompassing object
  - Consider a data structure that keeps a collection of data sorted in some way
    - Now consider returning a reference to an individual object with in the collection
    - How about return an actual array of sorted data

The this Reference

- Often in instance methods, you are accessing both instance variables and method variables
- If a method variable has the same name as an instance variable, updates will change the method variable, **NOT** the instance variable.

```
public class aClass
{
    private int counter;

    public int aMethod(...)
    {
        int counter;
        counter = 5;    // Which counter???
    }
}
```

- this is a pseudo-instance variable that is a **self-reference** to an object
  - It allows disambiguation between instance variables and method variable

The this Reference

- Often in instance methods, you are accessing both instance variables and method variables

```
public class aClass {
    public int counter;

    public aClass {...};

    public int aMethod(...)
    {
        int counter;
        counter = 5;        // method variable
        this.counter = 6;   // instance variable
    }
}

public class anotherClass {
    public static void main(String[] args)
    {
        aClass c = new aClass(...);
        System.out.println(c.counter); // instance variable
    }
}
```

The this Reference

- We also use the this reference with constructor:

```
public class Disk
{
    private int diskSize;
    private char diskChar;
    private char poleChar;

    public Disk(int aDiskSize, char aDiskChar, char aPoleChar)
    {
        diskSize = aDiskSize;
        diskChar = aDiskChar;
        poleChar = aPoleChar;
    }

    public Disk(int aDiskSize)
    {
        this(aDiskSize, '*', '|');
    }
    :
}
```

Garbage Collection

- When a reference to an object is reassigned, the original object can no longer be accessed through that reference
- If there is no other reference to that object, then it cannot be accessed
- In this case, the object has become garbage
  - An object sitting in memory that can no longer be an active part of the program
- If a program produces a lot of garbage, it can consume a lot of memory
- The garbage collector runs when needed to deallocate the memory taken up by garbage so that it can be reused
- The details of how it works are very interesting, but beyond the scope of this course

Wrappers

- Much useful Java functionality relies on Classes/Objects
  - Inheritance (Chapter 10)
  - Polymorphism access (Chapter 10)
  - Interfaces (Chapter 10)
- Predefine classes also help with objects
  - `ArrayList` can help if you want an array of **objects**
- Unfortunately, the Java primitive types are NOT classes, and thus cannot be used in this way

Wrappers

- **Wrapper classes** allow us to get around this problem:
  - Wrappers are classes that **wrap** objects around primitive values, thus makeing them compatible with other Java classes
    - We cannot store an `int` in an array of Object, but we could store an `Integer`
  - Each Java primitive type has a corresponding wrapper
    - `Integer`, `Float`, `Double`, `Boolean`, ...
  - Example:

```
Integer i, j, k;
i = new Integer(5);
j = new Integer(12);
k = new Integer(-24);
```

Wrappers

- Wrapper classes also provide extra useful functionality for these types:
  - Recall the `Integer.parseInt()` method is a static method that enables us to convert from a `String` to an `int`
  - Example: `Character.isLetter()` is a static method that tests if a letter is a character or not
- See more in API

Wrappers and Casting

- However, arithmetic operations are not defined for wrapper classes
  - So, if we want to do any **math** without our wrappers, we need to get the underlying primitive values
  - If we want to keep the wrapper, we then have to wrap the result back up
  - Logically, to perform `k = i + j;` where `i`, `j`, and `k` are variables of type `Integer` (class wrapper), we have to perform the following:

```
k = new Integer(i.intValue() + j.intValue());
```

    Get the primitive value of each `Integer` object, add them, and create a new `Integer` object with the result.

Wrappers

- In Java 1.4 and before:
  - Programmer had to do the ocnversions explicitly
- In Java 1.5 **autoboxing** was added
  - This does the conversion back and forth automatically
  - Saves the programmer some keystrokes
  - However, the work **still is done**, so from an efficient point of view, we are not saving
  - Should not sue unless absolutely necessary
- We will see more how wrappers are useful after we discuss inheritance, polymorphism, and interfaces

Parsing Primitive Types

- One ability of the Wrapper classes is static methods to parse strings into the correct primitive values
  - Example: `Integer.parseInt()`, `Double.parseDouble()`, `Boolean.parseBoolean()`
  - These enable us to read data in as Strings, then convert to the appropriate primitive type afterward
  - "12345" in a file is simply 5 ASCII characters (45, 50, 51, 52, 53)
  - To convert it into an actual `int` requires processing the characters
    - Let's now see the actual algorithm

Parsing Primitive Types

- We know that the character 0 ('0') is ASCII number 48
- Thus, our integer value is

$$(49-48)\times10^4+(50-48)\times10^3+(51-48)\times10^2+(52-48)\times10^1+(53-48)\times10^0$$

- This can be done manually in a nice efficient way using a simple loop
  - This is what `Integer.parseInt()` method does
  - See `MyInteger.java` and `Wrappers.java`

The Character Class

- The `Character` wrapper class provides many useful methods:
  - Example: case conversion, checking for letters, checking for digits
  - Can be useful when we are parsing text files ourselves
- The `String` class has some very useful methods as wel
  - See textbook for a lot of them and check `Stringy.java`

Inheritance

- Sometimes we want to build a new class that is largely like one we already have
  - Much of the functionality we need is already there, but somethings need to be added or changed
- We can achieve this in Object-Oriented language using **inheritance**
  - Attributes of a base class, or superclass are passed on to a subclass

Inheritance (is a)

- We can understand this better by considering the **"is a"** idea
  - A subclass object **is a** superclass object
  - However, some extra instance variables and methods may have been added and some other methods may have been change.
- Note that **"is a"** is a one way operation
  - Subclass "is a" superclass (specific "is a" general)
    - with some modifications or additions
  - Superclass is NOT a subclass (general not "is a" specific)
    - missing some properties

Inheritance (Real World) Example

- A car is a vehicle
- A motorcycle is a vehicle
- Honda Accord is a Car

Extending a Class

- Inheritance in Java is implemented by **extending** a class

```
public class NewClass extends OldClass
{
    :
}
```

- We then continue the definition of the `NewClass` as usual
- However, implicit in `NewClass` are all data and operations associated with the `OldClass`
  - Eventhough we do not see them in the definition, THEY ARE THERE!!!!

private, public, and protected

- We already know what `private` and `public` declarations mean
- The `protected` declaration is between `public` and `private`
  - Protected data and methods are directly accessible in the base class and in any subclasses (and in the current package)
  - However, they are not directly accessible anywhere else
- Note that `private` declaration are still part of subclass, but they are not accessible from the subclass point-of-view
  - See `SuperClass.java`, `SubClass.java`, `Subby.java`, and `ex15.java`

Inheritance Example

- Let's look at another example:
  - `MixedNumber` vs `MixedNumber2` classes
  - Both of them utilize `RationalNumber` class.
  - Both have the same functionality, but `MixedNumber` uses **composition** and `MixedNumber2` uses **inheritance**
  - Notes:
    - Simplicity of `MixedNumber2`
    - Read comments carefully
    - See `ex16.java`, `RationalNumber.java`, `MixedNumber.java`, `MixedNumber2.java`

Inheritance Example

- Let's look at line 67 to 109 in `ex16.java`
  - In this code, we are using `RationalNumber` references to access both `RationalNumber` and `MixedNumber2` objects
    - This is legal due to the "is a" relationship of `MixedNumber2` and `RationalNumber`
    - Recall that `MixedNumber2` **is a** `RationalNumber`
  - We will examine this type of access in more detail soon when we discuss polymorphism
- Important Note
  - When a **superclass reference** is used to access a **subclass object**
    - The **only mehtods that are callable** are those that were **initially defined in the superclass**
    - The object may have additional methods (defined initially in the subclass) that cannot be called.

Java Class History

- In Java, class `Object` is the base class of other class
  - This is the **mother of all classes** in Java
- In Java:
  - We do not explicitly say **extends** in a new class definition, it implicitly extends the `Object` class
  - The tree of classes that extend from the `Object` class and all of its subclasses is called the class hierarchy
  - All classes eventually lead back up to the `Object` class
  - This will enable consistent access of objects of different classes, as we shall see shortly

Polymorphism

- Idea of **Polymorphism**
  - Let's so some search
- Generally, it allows us to mix methods and objects of different types in a consistent way
- Earlier in the text, one type of polymorphism was already introduced
  - This is called **ad hoc polymorphism** or **method overloading**
    - In this case, different methods within the same class or in a common hierarchy **share the same name** but have different method signature (parameter + name)

```
public static int max(int x, int y) {...}
public static int max(int[] x) {...}
public static int max(int x, int y, int z) {...}
```

    - When a method is called, the call signature is matched to the correct method version
      - This is done during program **compilation**

Method Overloading

- If an exact signature match is not possible, the one that is **closest via "widening"** of the values is used
  - **Widening** means taht values of "smaller" types are cast to values of "larger" types
    - Example: `int` to `long`, `int` to `float`, or `float` to `double`
    - Fewer widenings provides a "closer" match
  - If two or more versions of the method are possible with the same amount of "widening", the call is **ambiguous**, and a compilation error will result
  - See `ex17.java`
  - Note: This type of polymorphism is not necessarily object-oriented.
    - Can be done in non-object-oriented languages

Subclass Polymorphism

- **Subclass polymorphism** sometimes called **true polymorphism**
- Consists of two ideas:
  - **Method overriding**:
    - A method defined in a **superclass** is redefined in a **subclass** with an **identical method signature**
    - Since the signatures are identical, rather than overloading the method, it is instead **overriding the method**

      - For subclass objects, the definition in the subclass replaces the version in the superclass

      - See `OverrideDemo.java`, `MyAList.java`, and `SortAList.java`

Subclass Polymorphism

- Polymorphism (continue)
  - **Dynamic (or late) binding**
    - The code executed for a method call is associated with the call during **run-time**
    - THe actual method executed is determined by the **type of teh object**, not the type of the reference
  - This allows superclass and subclass objects to be accessed in a regular, consistent way
    - Array or collection of superclass references can be used to access a mixture of superclass and subclass objects
    - This is very useful if we want access collections of mixed data types (example: draw different graphical objects using the same `draw()` method call for each)

Polymorphism

- Each subclass overrides the `move()` method in its own way

```
Animal[] a = new Animal[3];
a[0] = new Bird();
a[1] = new Person();
a[2] = new Fish();
for(int i = 0; i < a.length; i++)
    a[i].move();
```

  - References are all the same (`Animal`) but objects are not
  - The `move()` method invoked is that associated with the **object**, not the reference

Object, Method, and Instance Variable Access

- When mixed objects of difference classes, some access rules are important to know:
  - **Superclass references** can always be used to **access subclass objects**, but NOT vice versa

```
Animal a = new Bird();      // This is okay
Bird b = new Animal();      // NOT okay
```

  - Given a **reference R** of **class C**, only methods and instance variables that are defined (initially) in **class C or ABOVE** in the class hierarchy can be **accessed through R**
    - They still exist if defined in a subclass, but they are not accessible through R.

Object, Method, and Instance Variable Access

- Example
  - Suppose class `Fish` contains a new instance variable `waterType` and a new method `getWaterType()`

```
Fish f = new Fish();          // okay as usual
Animal a = new Fish();        // okay since Fish is an Animal
System.out.println(f.getWaterType());         // okay
System.out.println(a.getWaterType());         // NOT okay
System.out.println((Fish a).getWaterType()); // okay
                                              // (see below)
```

    - The above is NOT legal, even though the method exists for class `Fish`. The reason is that the method is not visible from the reference's point of view (`a` is an `Animal` reference so it can only "see" the data and methods defined in class `Animal`)
    - The last line is okay since we have now cast the reference to the `Fish` type, which CAN access the method

Object, Method, and Instance Variable Access

- Note that we can access these methods or instance variables **indirectly** if an overridden method access them
  - For example, if the `move()` method as defined in the class `Fish` called the `getWatherType()` method, and we called

```
Animal a = new Fish();
a.move();
```

    it would work fine
- Also note that if we cast a reference to a different type, and the object is not that type (or a subtype), we will get **ClassCastException**
  - If unsure, test using the `instanceOf` operator before casting
- See `ex18.java` for an example

Object, Method, and Instance Variable Access

- To summerize:
  - **Superclass reference CAN BE** used to reference **subclass objects**
  - **Subclass reference CANNOT BE** used to reference **superclass objects**
  - The **type of reference** determine what public data and methods are **accessible / can be seen**
  - The **type of the object** determines what data and methods **EXISTS**
    - Methods and data **initially defined within a subclass CANNOT BE** accessed via a **superclass reference**
    - The **type of the object** also determines which **VERSION** of an **overridden method** is called.

Abstract Classes

- Sometimes in a class hierarchy, a class may be defined simply to give cohesion to its subclasses
  - No objects of that class will ever be defined
  - But instance data and methods will still be inherited by all subclasses
- This is an **abstract** class
  - The keyword **abstract** used in declaration
  - One or more methods may be declared to be abstract and are thus not implemented
  - No objects may be instantiated

Abstract Classes

- Subclasses of an abstract class must **implement** all abstract methods, or they too must be declared abstract
- Advantages:
  - Can still use superclass reference to access all subclass objects in polymorphic way
    - However, we need to declare the methods we will need in the superclass, even if they are abstract
  - No need to specifically define common data and methods for each subclass – it is inherited
  - Helps to organize class hierarchy
- See `ex19.java`

Java Interface

- Java allows only **single inheritance**
  - A new class can be a subclass of only one parent (super) class
  - There are several reasons for this, from both the implementation (i.e. how to do it in the compiler and interpreter) point of view and the programmer (i.e. how to use it effectively) point of view
  - However, it is sometimes useful to be able to access an object through more than one superclass reference

Java Interface

- We may want to identify an object in multiple ways:
  - One **based on its inherent nature** (i.e. its inheritance chain)
    - For example, a `Person`
    - **Classes** are used to identify objects in this way
  - Others, **based on what it is capable of doing**
    - A swimmer
    - A musician
    - A performer
    - Note that a `Person` can potentially do many things
      - They can also be identified by his/her ability
    - **Interface** are used to identify objects in this way

Java Interface

- Java **interface** is a named set of **methods**
  - However, no method bodies are given (**just the header**)
  - Static constants are allowed, but no instance variables are allowed
- Any Java class (no matter what its inheritance) can implement an interface by implementing the methods defined in it
  - Essentially an **interface is stating an ABILITY of the class**
- **A given class can implement any number of interfaces**

Interface Examples

- Example:

```
public interface Laughable
{
    public void laugh();
}
```

- Example:

```
public interface Boolable
{
    public void boo();
}
```

- Any Java class can implement `Laughable` by implementing the method `laung()`
- Any Java class can implement `Boolable` by implementing the method `boo()`

Interface Examples

- Example:

```
public class Comedian implements Laughable, Boolable
{
    // Instance Variable(s)
    // Constructor(s)
    // Method(s)

    public void laugh()
    {
        System.out.println("Ha Ha Ha");
    }

    public void boo()
    {
        System.out.printon("Boooooooooo.....");
    }
}
```

Java Interface

- An interface variable can be used to reference any object that implements that interface
  - Note that the same method name (ex: `laung()` below) may in fact represent different code segments in different classes
  - **But only the interface methods are accessible through the interface reference**
  - Thus, even though a single class may implement many interfaces, if it is being accessed through an interface variable, the methods in the other interfaces are not available.
    - The interface masks the object such that only the interface methods are visible or callable
    - If other methods are attempted to be accessed, a compilation error will result

Example

- Example:

```
Laughable[] funnly = new Laughable[3];
funny[0] = new Comedian();
funny[1] = new SitCom();  // Assume to implement Laughable
funny[2] = new Clown();   // Assume to implement Laughable

for(int i = 0; i < funny.length; i++)
    funny[i].laugh();

funny[0].boo();           // Compilation error
```

- See `ex20.java`
- This restricted access is the **same behavior** that we discussed about **SuperClass** reference to **SubClass** object

Example

- Recall our previous discussion of polymorphism
- This behavior also applies to interfaces – the interface acts as a superclass and the implementing classes implement the actual methods however they want
- **An interface variable can be used to reference any object that implements that interface**
  - However, only the interface methods are accessible through the interface reference
- Recall previous example (similar to polymorphic behavior – the `Animal` hierarchy):

```
Laughable[] funnly = new Laughable[3];
funny[0] = new Comedian();
funny[1] = new SitCom();  // Assume to implement Laughable
funny[2] = new Clown();   // Assume to implement Laughable

for(int i = 0; i < funny.length; i++)
    funny[i].laugh();

funny[0].boo();           // Compilation error
```

Generic Operations

- How does it benefit us to be able to access objects through interfaces?
  - Sometimes, we are only concerned about a given property of a class
    - The other attributes are methods still exist, but we do not care about them for what we want to do
  - For example, **Sorting**
    - We can sort a lot of different types of objects
      - Various numbers
      - People based on their names alphabetically
      - Movies based on their titles
      - Employees based on their salaries
    - Each of these classes can be very different
    - However, something about them all allows them to be sorted

Generic Operations

- They all can be compared to each other
  - So, we need some method that invokes this comparison
- In order to sort them, we do not need to know or access anything else about any of the classes
  - Thus, if they all implement an interface that defines the comparison, we can soft them all with a single method that is defined in therms of that interface
- For better understanding, let's see an example

Generic Operations

- Consider the `Comparable` interface

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

- The `compareTo()` method returns:
  - negative number if the current object is less than `o`
  - 0 if the current object is equals to `o`
  - positive number if the current object is greater than `o`
- What is `T`?
  - `T` is located where a primitive type or a class should be
  - It represents a class (can be any class) that you define in <T>
  - For now, an actual class must be used in place of `T`

Generic Operations

- Example

```
public class Person implements comparable<Person>
{
    // instance variable(s)
    // constructor(s)
    // method(s)
    int compareTo(Person o);
}
```

- Note that `T` is replaced by `Person`
  - This enforce that the object of class `Person` will have the `compareTo()` method that can be used to compare with **another** `Person`
    - Or object of another class that inherits `Person` (discuss)
    - Prevent a user to compare a `Person` with a `Bird`

Generic Operations

- Now consider what we need to know to sort data:
  - Is `array[i]` less than, equal to, or greater than `array[j]`
- Thus, **we can sort `Comparable` data without knowing anything else about it**
  - Awesome! Polymorphism allows this to work
- Think of the objects we want to sort as **black boxes**
  - We know we can compare them because they implement `Comparable`
  - We do not know (or need to know) anything else about them
- Thus, a **single sort method will work for an array of any `Comparable` class**
  - Let's write it now, altering the code we already know from our simple sort method
  - See `SortAll.java` and `ex21.java`
    - Also see `SortAllT.java` and `ex21T.java`