Inheritance and Polymorphism

Introduction to Inheritance

- Problem: we need to write a program (software) to solve a new problem (How to re-sue existing software to build more complex software prior to ~1995, invention of OOP)
  - We want to write a `NewClass` class to solve the problem
  - We find a program (`SomeClass`) that can be used as the starting point to build our more complex software.
  - We make a copy of the program.
  - Then, we make changes to the copy of software, so the new software does what we want.
  - However, we have the problem of:
    - **Redundancy**: we can have multiple copies of the same method
    - **Hard to maintain programs**: when we update the original software (from which we made the new software), we may need to update our programs that are based on the existing software
- After the invention of OOP, we define (not copying!) the new class to **inherit** from the original class.
  - The new class will **inherit**(=receive) all the variables and (normal) methods from an existing class
  - Example:

```
public class SomeClass{
  public int x;
  public SomeClass{
      x = 99;
  }
  public void method1() {
      System.out.println("I am SomeClass.method1(). x = " + x);
  }
  public void method2 () {
      System.out.println("I am Someclass.method2(). x = " + x);
  }
}
```

```
public class NewClass extends SomeClass{
  NewClass() { }
  // No other methods defined
}


public static void main(String[] args) {
  NewClass b = new NewClass();
  b.method1(); // Invokes SomeClass.method1()
  b.method2(); // Invokes SomeClass.method2()
}
```

    - If An inherited method is not appropriate (does not do what we want), we **replace (override)** that method with a new method with the same signature.
        - Methods defined inside the `NewClass` will take priority over an inherited method with the same method signature (this mechanism is called **overriding**).
    - If original class does not have a suitable method for some task in the new class, we can add new methods to our `NewClass` to perform that task.
        - These new methods will only be defined in the `NewClass` (and will not be defined or inherit in the original class).
    -
        - See 02-override
- Accessing an overridden method and an overriding method:
    - Notice there are two different methods named `method1()` with the same signature if we override the method.
        - The original `method1()` is in `SomeClasses` (the overridden method)
        - The new `method1()` is in `NewClass` (the overriding method)
        - When writing methods in `NewClass` , both methods are available for use (=accessible).
    - To access the overridden method, we use the `super` keyword, which always refers to the members in the super class.
        - For example, `super.method1()` refers to the `method1()` defined in `SomeClass`
- Important note: Accessibility modifiers are enforced on inherited members.
    - Note: the subclass and its superclass are separate class.
    - Therefore, methods defined (written) inside a subclass cannot access `private` members in the super class.
    - Only the **unmodified inherited methods** in the subclass can access the inherited `private` members.

Object-Oriented Thinking

- Object-Oriented Thinking
  - In order to maximize the inheritance mechanism to re-use existing software, we need to adopt the Object-Oriented Design methodology when developing the classes.
  - The Object-Oriented Design methodology organizes object class in a hierarchy according to common properties and actions
  - The Object-Oriented methodology can minimize the re-use of variables and methods.
- How to maximize the sharing of properties and/or actions among classes
  - We use a class to model objects of the same type
  - Different classes can have common properties and/or behaviors
  - To maximize sharing of common properties/behaviors, we generalize different classes into a large (super) class.
    - The `is-a` generalization method will give us the maximum sharing of properties and actions
- The Object-Oriented Design methodology uses the **is-a** generalization technique to achieve maximal sharing of properties and actions between classes.
- How to design the class hierarchy using the is-a generalization technique
  - First, determine all the program classes that we will need to solve the problem. Determine the properties and actions that are needed in each class.
  - Then, generalize similar program classes using the is-a generalization. Use the properties and actions in each class to find the more general classes.
  - If possible, generalize further. Use the final hierarchy to determine the properties and actions of each class.

> **Superclass**: the more general class in the `is-a` relationship (aka, parent class)

- A super class has a number of properties and actions.

> **Subclass**: the more specific class in the `is-a` relationship (aka, child class)

- Every object of the subclass will have all the properties and actions in the superclass.
- In addition, the subclass object can have other properties and actions not found in the superclass.
  - See `TestGeometricObject.java`
- The OOP design allows us to avoid duplications of codes when solving a problem!
  - See `TestOldWay.java`
  - We define a superclass that contains the common (shared) properties and actions in all classes

        - Some methods in the superclass may not have a useful method body - it's OK. This is very useful for the polymorphism mechanism.
    - We create subclasses that extend the superclass.
        - For private instance variables, we must use its accessor/mutator methods to use the variables.
        - For public instance variables, we can access them directly.
    - We can override some methods in the subclass.
- Relationship between a subclass and its superclass
    - A subclass inherits all variables and (normal) methods from its superclass.
    - A subclass do not inherit any constructor method from its superclass.
        - A constructor in the subclass must invoke a constructor in the superclass
        - A subclass object always contains a superclass object.
        - Objects are initialized using its constructor
    - Rule: a constructor in the subclass must invoke some constructor in its superclass as its first statement.
        - The keyword `super(...)` is used to invoke a constructor in its superclass.
    - Rule: if a constructor in the subclass does not invoke any constructor in its superclass, then, the Java compiler will automatically insert the call `super()` as the first statement. That is, when the first statement in a constructor is not `super(...)`, the Java compiler will call the default constructor.

```
public class NewClass extends SomeClass{
  NewClass() {
    // Compile error -- why?
  }
}

public class SomeClass {
  public int x;
  public SomeClass(int a) {
    x = a;
  }
}
```

    - The compile error is because the constructor `NewClass()` does not contain any `super(...)` calls, so Java compile will insert `super(...)`:

```
public class NewClass extends SomeClass{
  NewClass() {
    super();
  }
}
```

    - However, there is not matching constructor (`SomeClass()`) defined in the supercalss, which causes the error.
- Consequences of the constructor invocation rule in Java:
    - Constructor invocation rule in Java:
        - If a class `B` inherits from class `A` , then every constructor in class `B` must invoke some constructor in class `A` .
    - Consequence:
        - If another class `C` inherits from the class `B` , then every constructor in class `C` must invoke some constructor in class `B` and in class `A` .
        - This phenomenon is called **constructor chaining**.
- Summary:
    - A subclass inherits all normal members (including `private` members) from its superclass
    - Methods in the subclass cannot access the `private` inherited members directly.
    - A subclass object contains (all members in) a supercalss object.
    - A subclass do not inherit any constructors from its superclass.
    - Because a subclass object contains (All members in) a superclass object, every constructor in the subclass must invoke a constructor in the superclass.
- Sometimes, we must use the overridden method inside the super class (i.e., use `super.methodName()`)
    - See `05-bank-account`

Overloading vs. Overriding

- **Overloading**: defining different methods with the same (method) name but with different (method) signatures

```java
public class SomeClass {
  public int x;
  public SomeClass (int a) {
    x = a;
  }
  public void method1() {
    System.out.println("SomeClass.method1()");
  }
  public void method2() {
    System.out.println("SomeClass.method2()");
  }
}


public class NewClass extends SomeClass {
  NewClass (int a) {
    super(a);
  }
  // Inherits: method1() and method2()
  public void method1(int a) { // Overloads method1()
    System.out.println("NewClass.m1(int)");
  }
  public void method3() {
    method1(); // Invokes SomeClass method1
    method1(22); // Invokes NewClass method1(int)
  }
}
```

- **Overriding**: replacing an inherited method by defining a method with the same (method) signature

```
public class NewClass extends SomeClass {
  NewClass(int a) {
    super(a);
  }
  // Inherites: method1() and method2()
  public void method1() { // overrides method1()
    System.out.println("NewClass.m1()");
  }
  public void method3() {
    method1(); // Invokes NewClass method1
    super.method1(); // Invokes SomeClass method1
  }
}
```

- In Java, we can add the special override annotation `@Override` before an overriding method for clarity:

  ```
  public class NewClass extends SomeClass {
    // Inherits: method1(double)
    @Override
    public void method1(double x) {
      System.out.println("x = " + x);
    }
  }
  ```

  - Java compiler will report an error if the defined method does not override any inherited methods.
- Additional conditions on overriding methods
  - The overriding method must have the same return type as the overridden method. We will get a type incompatible error when we use different return types.
  - The overriding method should have the same accessibility modifier as the overridden method. Complicated errors can result when you do not use the same accessibility.

Things that the Java Compiler does automatically

- Every object must be initialized; therefore
  - Every class must have a constructor method
  - if a class does not have one, the Java compiler will insert the default constructor
- Every subclass object contains a superclass object (that must be initialized); therefore
  - A subclass constructor must invoke `super()` as its first statement.

    - If not so, the Java compiler will insert `super();` at the beginning.
- Every class in Java is descended from one special class called the `Object` class. i.e., the `Object` class is the parent class of every class in any Java program
    - If no inheritance is specified when a class is defined, the Java compiler will automatically insert `extends Object` in the class definition.
    - Every class in Java inherits from the `Object` class.
    - Every object in Java will have all the methods defined in the `Object` class.
- One important method in the `Object` class is `toString()`
    - `toString()` returns a string representation of this object.
    - Furthermore, the `System.out.println()` method invokes the `toString()` method so it can print out any object.
    - Therefore, we can control the print out of an object by overriding the `toString()` in a class.

Introduction to Polymorphism

- So far, we have always used a reference variable to point to an object of the same class. Now, we will use a reference variable to point to an object of a different class.
    - Example:

```
public class SomeClass {
  public int x = 44;
  public void method1() {
    System.out.println("SomeClass.m1()");
  }
  public void method2() {
    System.out.println("SomeClass.m2()");
  }
}
```

    - Then, the following are legal
        - Define a reference variable: `SomeClass a;`
        - Create a SomeClass object: `a = new SomeClass();`
        - `a.x`
        - `a.method1()`
        - `a.method2()`
- The Correctness of Program Execution (Suppose a `SomeClass` variable `a` is referencing to some arbitrary object with `a.x` , `a.method1()` , and `a.method2()` defined)
    - Rule: a reference variable must refer to an object that contains all members in a legal request

        - Reason: the referred object must perform all acrtions in the class or else we can have a request error.
        - The simplest way to satisfy this rule is to refer to an object of the same class.
        - However, there is another safe (=correct) way due to the inheritance relationship.
    - A subclass object can perform all actions that a superclass object performs.
        - Therefore, it is **safe** to use a superclass reference variable to request members in a subclass object.
        - Java allows we to access members in a subclass object using a superclass reference variable.
- Dynamic Dispatch (aka: late binding)
    - The request `a.method()` will execute the method in the object that `a` is currently pointing to.
    - This feature is called **dynamic dispatch** or **late binding**, which means decision on which method to run is at the last moment.
    - Example:

```
public class SomeClass {
  public int x = 44;
  public void method1() {
    System.out.println("SomeClass.m1()");
  }
  public void method2() {
    System.out.println("SomeClass.m2()");
  }
}
public class NewClass extends SomeClass {
  @Override
  public void method1() {
    System.out.println("NewClass.m1()");
  }
  public void method3() {
    System.out.println("newClass.m3()");
  }
}
public class myProg {
  public class void main(String[] args) {
    SomeClass a = new NewClass(); // allowed!
    System.out.println(a.x);
    a.method1(); // Invokes method1() in NewClass
    a.method2();
    a.method3(); // illegal
  }
}
```

- Because `a` is pointing to a `NewClass` object, `a.method1()` will execute `NewClass` 's `method1()` even though `a` is a `SomeClass` object.

- Polymorphism
  - Consider the following program:

```
SomeClass a; // superclass reference variable
// Use superclass variable to access overridden method in superclass
a = new SomeClass(); // refer superclass object
a.method1(); // calls SomeClass.method1()
// Use superclass variable to access overridden method in subclass
a = new NewClass(); // refers subclass object
a.method1(); // class NewClass.method1()
```

  - The same expression `a.method1()` invokes different methods

> **Polymorphism**: the phenomenon that the same expression (program code) can result in different actions. Polymorphism is caused by using a superclass reference variable to access overridden members in the superclass and their subclasses and late binding.

- A reverse case: a superclass object may perform fewer actions than a subclass object.
  - Therefore, it is illegal to use a subclass reference variable to access members in a superclass object.

Application of Polymorphism: Selection Sort Algorithm

> **Sorting an array** = re-arrange the values in an array so that the values are ordered.

- In our discussion, we will sort the array in ascending order.
- There are many array sorting algorithms, but we will first examine the **selection sort algorithm**.
  - Selection sort finds the smallest number in the list and swaps it with the first element.
  - It then finds the smallest number remaining and swaps it with the second element.
  - And so on, until only a single number remains (i.e., the last number in the list)

```
public static void selectionSort(int[] list) {
  for (int i = 0; i < list.length-1; i++) {
    // Find the minimum in the list[i...list.length-1]
    int min      = list[i]; // Assume the first element is min
    int minIndex = i;       // Index where min is found
    for (int k = minIndex+1; k < list.lenght; k++) {
      if (list[k] < min) {  // Find a smaller element
        min      = list[k]; // Update min value
        minIndex = k;       // Update its index
      }
    }
    // Swap list[i] with list[minIndex] if necessary
    if (minIndex != i) {
      // Swap list[minIndex] and list[i]
      // Standard exchange alg
      int help       = list[minIndex];
      list[minIndex] = list[i];
      list[i]        = help;
    }
  }
}
```

- Due to polymorphism, we can change the selection sort algorithm for integers to `Circle`, `Rectangle`, and even `GeometricObject` objects.
  - In order to do so, the superclass object must provide all the necessary actions used in the `selectionSort()` algorithm.
  - To be more specific, that is why we have defined `getArea() { return 0 }` in the `GeometricObject` previously.

```
public static void selectionSort(GeometricObject[] list) {
  for (int i = 0; i < list.length-1; i++) {
    GeometricObject min = list[i];
    int minIndex        = i;
    for (int k = minIndex+1; k < list.lenght; k++) {
      if (list[k].getArea() < min.getArea()) {
        min      = list[k];
        minIndex = k;
      }
    }
    if (minIndex != i) {
      GeometricObject help = list[minIndex];
      list[minIndex]       = list[i];
      list[i]              = help;
    }
  }
}
public static void mian(String[] args) {
  GeometricObject[] myList = new GeometricObject[4];
  myList[0] = new Circle("red", 2);
  myList[1] = new Rectangle("blue", 1, 1);
  myList[2] = new Circle("white", 5);
  myList[3] = new Rectangle("black", 4, 4);
  selectionSort(myList)l
  for (int i = 0; i < myList.length; i++) {
    System.out.println(myList[i]);
  }
}
```

- Summary
  - A common technique to generalize code is to write methods with a superclass type as parameter type.
  - Such a method can receive objects of any subclass type as argument.
  - Requirement:

        - The superclass type must provide all the actions necessary to code the method
    - Java provides an interface mechanism that is similar to the inheritance mechanism for defining superclass type and subclass type relationship.
    - Using this interface mechanism, we can make a superclass type that can unite the `String`, the `Circle`, and the `Rectangle` classes

Exception Handling and Polymorphism

- The exception types in Java are also organized as an inheritance hierarchy. The root type is `Exception`
- We can catch more general exceptions using types higher up in the exception hierarchy.

```
public static void main(String[] args) {
  int[] a = new int[10];
  try {
    a[99] = 1;
  } catch (Exception e) { // higher up.
    System.out.println(e);
  }
}
```

    - Use `Exception`, we will catch all types of exceptions
- We can use multiple `catch` clauses, but the more specific exceptions should come first.

```
public static void main(String[] args) {
  int[] a = new int[10];
  try {
    if (Math.random() < 0.5) {
      a = null;
    }
    a[99] = 1;
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(e);
  } catch (Exception e() {
    System.out.println(e);
  }
}
```

Upcasting, Downcasting, and `instanceof` operator

> **Upcasting**: casting (converting) a subclass reference into a superclass reference.

- Upcasting is a safe operation because we can make fewer requests using a superclass variable than using a subclass variable.

> **Downcasting**: casting (converting) a superclass reference into a subclass reference.

- Downcasting is an unsafe operation because we may make an illegal request using a subclass variable.
- Under some situations, downcasting can be safe and necessary:
  - We first perform an upcasting operation
  - Later, we want to perform an action defined in the subclass
- To perform a safe downcasting operation:

```
public static void main(String[] args) {
  GeometricObject a = new Circle("red", 1); // Upcasting
  System.out.println(a.getArea()); // safe
  // System.out.println(a.getRadius()); // illegal -- getRadius() is not defined in
  Circle b;
  b = (Circle) a; // Explicit upcasting
  System.out.println(b.getRadius()); // allowed
}
```

- We can also write similar codes for downcasting a `GeometricObject` object to a `Rectangle` object. However, how can we write a program to cast the superclass variable references to a subclass variable of non-specific subclass?
  - The `instanceof` boolean condition:

    ```
    objectRefVar instanceof className;
    // returns true if the object referred to by objectRefVar
    // is an object of className type or a subclass type of
    // className. Otherwise, returns false
    ```

  - Using the `instanceof`, we can solve the problem:

```
public static void main(String[] args) {
  GeometricObject a;
  if (Math.random() < 0.5) {
    a = new Circle("red", 1);
  }
  a = new Rectangle("blue", 2, 1);

  if (a instanceof Circle) {
    Circle b = (Circle) a; // downcast to a circle
    System.out.println(b.getRadius();)
  } else if (a instanceof Rectangle) {
    Circle b = (Rectangle) a; // downcast to a rectangle
    Syste.out.println(b.getWidth());
    Syste.out.println(b.getHeight());
  } else {
    System.out.println("Invalid subclass type");
  }
}
```

The `protected` Modifier and the `final` Modifier

- The accessibility modifiers indicates the degree of trust (closeness) of program code written by different people.
  - Highest level of trust: Code inside a class
  - 2nd highest level of trust: code inside a package
  - Lowest level of trust: code inside a different package.
- The `protected` modifier will allow subclasses inside a different package to access data fields or methods in the superclass.
  - Syntax to define a member with `protected` accessibility:

    ```
    protected memberDefinition;
    ```

  - Where we can access with `protected` accessibility:
    - from inside a method in the same class (closest association)
    - from inside a method in the same package (2nd closest association)
    - from inside a method in a subclass defined outside the package
    - but not from inside a method in an unrelated class defined outside the package

Modifier on Members	from the same class	from the same package	from subclass in different package	from a different package
`public`	OK	OK	OK	OK
`protectd`	OK	OK	OK	No
`default`	OK	OK	No	No
`private`	OK	No	No	No

- - See `TestNewCircle.java`
- A class with the `final` qualifier cannot be extended (i.e., used as a superclass)

```
// This class cannot be extended
public final class myClass {
  // data fieds, constrcutors, and methods omitted.
}
```

- A method with the `final` qualifier cannot be overridden in a subclass

```
public class myClass {
  // This method cannot be overridden
  public final void method1() {
    // Do something,
  }
}
```

- - See `FinalCircle.java` and `FinalGeometricObject.java`

Hiding Variables and Multi-Inheritance

- If a subclass defines a variable `x` with the same name as its superclass:
  - the name `x` will refer to the variable in the subclass:
    - The variable `x` in the subclass will overshadow (hide) the variable in the superclass.
  - The variable `x` in the superclass can be accessed in the subclass using `super.x` (or through a non-overridden method)
  - However, it's a terrible idea to override variables.
- Multiple-Inheritance

- Java allows a class to inherit from only one superclass, while other languages (such as C++) can inherit from multiple class and made things very complicated.
- Java does implement some features of multiple inheritance through **interface**:
  - A class can have multiple parent interfaces
  - But these parent interfaces must be completely empty (=no variables and contains only method declarations)