Inheritance

- Conceptual
- Is-A relationship compared to contains-a
- Terminology
- Overloading compared to Overriding
- super
- isInstanceOf and getClass()

Inheritance

- A crucial feature of object-oriented programming languages
  - One class (derived class, subclass, child class) is constructed …
  - … by including (extending, inheriting) information …
  - … from another (base class, superclass, parent class) …
  - … and adding new information / redefining existing
- Example
  - Base class: Clock
    - setTime
    - getTime
    - tick
  - Derived class: Alarm Clock
    - Same methods as Clock plus a few additional ones: setAlarm, ring
Can We Avoid Code Copying and therefore redundancy?

- Alarm Clock “IS-A” Clock
- Operations on Clock (e.g. setTime) should be inherited by Alarm Clock
- Alarm Clock should only have to add information specific to alarm clocks
  - setAlarm
  - ring
- Inheritance provides just this capability

Inheritance

- One class (derived class, subclass, child class) is constructed by including (extending, inheriting) information from another (base class, superclass, parent class) then also adding new information and/or redefining existing information
- To derive a class D from a base class B, use:
  
  ```java
  public class D extends B {
  ...
  }
  ```

- Example (we will look at this in next two slides):
  - **Base class**: public class Shape
  - **Derived class**: public class Circle extends Shape
- Derived class inherits all instance variables, methods from base class. It can also define new instance variables, methods
- **Polymorphism**: object in derived class can be used anywhere base class is expected (an alarmClock “is a” Clock!)
Inheritance vs. Composition

- **Inheritance:** a way to build new classes out of old ones
  - Objects in subclass inherit data, methods from superclass
  - Object in a subclass “is-a”(n) object in superclass
- **Association:** another way to build new classes out of old
  - Class definitions may include instance variables which are objects of other class types
  - Object in a new class “has-a”(n) object in the original class
- **Composition:** the strongest form of association – when the lifetime of the enclosed object is completely dependant on the lifetime of the object that contains it

Implements vs. Extends When Defining a Class

- **implements:**
  - Keyword followed by the name of an interface
  - Interfaces only have method prototypes
  - Can’t create an object of an interface type
  - Can have a reference of the interface type point to an object of the class that implements it
- **extends:**
  - Keyword followed by the name of a class
  - That class contains full method definitions
  - Can create objects of that base class type
  - Can have reference of the base class type point to an object of the class that extends it
Inheritance More Generally

- Classes / objects have a natural “is-a” hierarchy
- Object-oriented programming provides mechanisms for exploiting this for
  - Code re-use
    - Common operations implemented in super classes
  - Polymorphism
    - Objects in subclasses can be used wherever superclass objects are needed

Example: People at University

- Base class: person
- Derived classes: student, faculty, administrator
- Derived from those: undergrad, grad, instructor, professor,…
University Person Example

```
public class Person {
    String name;
    String idNum;

    Person( … ) [various]
    String getName( )
    String getIdNum( )
    void setName( String )
    void setIdNum( String )
    String toString( )
    boolean equals( Person )
}

public class Student extends Person {
    int admitYear;
    double gpa;

    Student( … ) [various]
    int getAdmitYear( )
    double getGpa( )
    void setAdmitYear( int )
    void setGpa( double )
    String toString( )
    boolean equals( Student )
}

public class Faculty extends Person {
    int hireYear;

    Faculty( … ) [various]
    int hireYear( )
    void setHireYear( int )
    String toString( )
    boolean equals( Faculty )
}
```

Method Overriding

- A derived class can define new instance variables and methods (e.g. `hireYear` and `getHireYear( )`).
- A derived class can also redefine (override) existing methods.

```java
public class Person {
    public String toString() { … }
}

public class Student extends Person {
    public String toString() { … }
}
```

```
Student bob =
    new Student("Bob Goodstudent","123-45-6789",2004,4.0 );
System.out.println( "Bob's info: " + bob.toString( ) );
```
Overriding vs. Overloading

- **Overriding**: a derived class defines a method with same name, parameters as base class
- **Overloading**: two or more methods have the same name, but different parameters

**Example**

```java
public class Person {  
    public void setName( String n ) { name = n; }
    ...
}

public class Faculty extends Person {  
    public void setName( String n ) { super.setName("The Evil Professor " + n ); }
    public void setName( String first, String last ) { super.setName( first + " " + last ); }
}
```

**Consider**:

```java
Faculty carol = new Faculty("Carol Tuffteacher","999-99-9999", 1995);
Person p = carol;
System.out.println( p.toString() );
```

**Which version of toString - Person or Faculty - is called?**

- **Early (static) binding**
  - `p` is declared to be of type `Person`
  - Therefore, the `Person` version of `toString` is used
- **Late (dynamic) binding**
  - The object to which `p` refers was created as `Faculty` object
  - Therefore, the `Faculty` version of `toString` is used

**Java uses late binding (C++ by default uses early binding)**

- Early binding is more runtime efficient (decisions about method versions can be made at compile time)
- Late binding respects encapsulation (object defines its operations when it is created)
Polymorphism

- **Java’s late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms).
- **Example**: Create an array of various university people and print.

```java
Person[] list = new Person[3];
list[0] = new Person("Col. Mustard", "000-00-0000");
list[1] = new Student("Ms. Scarlet", "111-11-1111", 1998, 3.2);
list[2] = new Faculty("Prof. Plum", "222-22-2222", 1981);
for (int i = 0; i < list.length; i++)
    System.out.println(list[i].toString());
```

- **What type is list[i]?** It can be a reference to any object that is derived from Person. The appropriate toString will be called.

Example of Overloading/Overriding

```java
public class Base {
    public void m (int x) { ... }
}

public class Derived extends Base {
    public void m (int x) { ... }
    public int m (int x) { ... }
    public void m (double d) { ... }
}

// The following appears in the same package as above
Base b = new Base( );
Base d = new Derived( );
Derived e = new Derived( );
```

- **Overloading**: with increased visibility
- **Overriding**: with decreased visibility
- **Error! duplicate method declaration**
- **Error! Since d is declared Base, the compiler looks for Base:m(double) Doesn’t exist! So this does not make it past the compiler, even though Derived:m(double) is defined!**
Calling an overridden function

- Possible but use sparingly.
  - Overriding hides methods of the base class (can still access them using `super.methodName()` in subclass, but not in "outside world")

```java
public class Person {
    public String toString() { /*one def here*/
        ...
    }
}
```

```java
public class Administrator extends Person {
    public String toString() { /*different def here*/
        public String regPrint() {
            return super.toString(); /* will use Person's def of toString*/
            return toString(); /* will use Administrator's def of toString*/
        }
    }
}
```

- Often better to pick a different name rather than overload if you want both.

Derived class: Student

```java
package university;
public class Student extends Person {
    private int admitYear;
    private double gpa;
    public Student() {
        super();
        admitYear = -1;
        gpa = 0.0;
    }
    public Student(String n, String id, int yr, double g) {
        super(n, id);
        admitYear = yr;
        gpa = g;
    }
    public Student(Student s) {
        super(s);
        admitYear = s.admitYear;
        gpa = s.gpa;
    }
    //...other methods in part 2
}
```
Understanding the Student

- **extends** specifies that `Student` is subclass of `Person`:
  ```java
  public class Student extends Person
  ```
- **super**
  - When creating a new Student object, we need to initialize its base-class instance variables (from Person)
  - This is done by calling `super( ... )`. E.g.
    ```java
    super(name, id) invokes constructor Person(name, id)
    ```
  - `super( ... )` must be the first statement of your constructor
  - If you do not call `super()`, Java will automatically invoke the base class’s default constructor
  - If the base class’s default constructor is undefined? Error
  - You must use `super( ... ), not Person( ... )`

---

**super VS. this**

- **super**: refers to the base class
  - Can invoke any base class constructor using `super( ... )`
  - Can access data and methods in base class (Person) via `super`
    ```java
    E.g., toString( ), equals( ) invoke the corresponding methods from Person base class using super.toString( ) and super.equals( )
    ```
- **this**: refers to current class / object
  - Can refer to own data and methods using `this` (usually unnecessary)
  - Can invoke any of its own constructors using `this( ... )`. Like `super`:
    - Can only be done within a constructor
    - Must be the first statement of the constructor
    - Example
      ```java
      public Faculty(Faculty f) {
        this(f.getName( ), f.getIdNum( ), f.hireYear);
      }
      ```
Inheritance and private

- Student inherits all private data (name and idNum) from Person
- However, private members of base class cannot be accessed directly

```java
public class Student extends Person {
    ...
    public void someMethod() {
        name = "Mr. Foobar"; // Illegal!
    }

    public void someMethod2() {
        setName("Mr. Foobar"); // OK
    }
}
```

- Why?
  - Although Student inherits from Person ...
  - ... they are different classes

Public, Protected, Package(default) and Private

- Select which level of visibility

<table>
<thead>
<tr>
<th>Access Level/Group</th>
<th>Class</th>
<th>Package</th>
<th>SubClass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected (avoid)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>package (default)</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Shadowing

- Can we override instance variables just like methods?
- Yes, but be careful!
  - Overriding instance variable is called **shadowing**
  - Shadowing hides instance variables of base class (can still access them using `super.varName` in subclass, but not in "outside world")

```java
public class Person {
    String name;
    ...
}
public class Administrator extends Person {
    String name; // name refers to Administrator’s name
}
```

- Confusing! Better to pick a new variable name

Object

- Recall: inheritance induces "is-a" hierarchy on classes
  - Undergrad "is-a" Student
  - Student "is-a" Person
  - etc.
  - Person "is-a" ....?
  - Person "is-a"(n) **Object**
  - Student "is-a"(n) **Object**

```
Class Diagram:
```
```
More on Object

- Special class at top of class inheritance hierarchy
- Defined in `java.lang` (so available in every program)
- Every class is derived (either directly or indirectly) from `Object`
  - If a class is not derived from anything, it is automatically derived from `Object`
  - e.g.
    ```java
    public class Foo { ...}
    is equivalent to
    public class Foo extends Object {...}
    ```
- Structure of `Object`
  - No instance variables
  - A number of methods, including:
    - `toString()`
    - `equals (Object o)`
  - Note: parameter to `equals` has type `Object`, so any object can be an argument
  - These methods can (and usually should) be overridden

Class vs. Type Information

- In Java
  - Every object is in one class (the one it was created from using `new`)
  - Objects may have many types
    - Interfaces
    - Superclasses
  - E.g. consider
    ```java
    Student bob = new Student();
    Person p = bob;
    ```
  - Class of object pointed to by `bob`, `p` is `Student`
  - Type of object can be `Student`, `Person`, `Object`, etc.
Accessing Class and Type Information

- Objects can access their class info at run-time
  - `getClass()`
    - Method defined in `Object`
    - Returns representation of object’s class
    - E.g.
      ```java
      Person bob = new Person( ... );
      Person ted = new Student( ... );
      if ( bob.getClass() == ted.getClass() ) // false (ted is really a Student)
      ```
- `instanceof`
  - Java boolean operator (not a method)
  - Returns true if given object “is-a”(n) object of given (class) type
  - E.g.
    ```java
    Student carol = new Student ( ... );
    if (carol instanceof Person) // true, because carol “is-a” Person
    ```

Object Casting

- Recall casting in primitive types
  - Casting: conversion of elements from one type to another
  - Widening Conversion
    - Every element in source type is a element in destination type
    - Can be done automatically
      ```java
      double x = 3;  // 3 (int) widening conversion to double
      ```
  - Narrowing Conversion
    - Elements in source type are not necessarily elements in the destination type
    - Must use explicit type conversions to perform this casting
      ```java
      int x = (int)3.0; // 3.0 explicitly cast to int
      ```
- Similar notions can be found with object types also
  - Upcasting
    - Casting a reference to a superclass (casting up the inheritance tree)
    - Always done automatically and is always safe
    - Just ignore the parts that were added by the subclass
  - Downcasting
    - Casting a reference to a derived class
    - Requires explicit casting operator, which checks type info at run-time
    - Can cause runtime error
Safe Downcasting

- Illegal downcasting results in a thrown ClassCastException at run-time
- Q: Can we check for the legality of a cast before trying it?
- A: Yes, using instanceof

Example
- Given: ArrayList of university people
- Want: Print the GPAs of the students
- Solution approach
  - Iterate through list
  - Print GPAs only of Students

equals() Reconsidered

- Recall definition of equals()
  - ... in Person
    public boolean equals (Person p) {
      if (p == null) {
        return false;
      }
      return name.equals(p.getName()) &&
      idNum.equals(p.getIdNum());
    }
  
  - ... in Student
    public boolean equals( Student s ) {
      if (s == null) {
        return false;
      }
      return super.equals(s) &&
      admitYear == s.admitYear &&
      gpa == s.gpa;
    }
- What does following do?
  public static void main (String[] args) {
    Student bob = new Student ("R. Goode", "234-56-7890", 1998, 3.89);
    Faculty bob2 = new Faculty ("R. Goode", "234-56-7890", 2005);
    System.out.println (bob.equals (bob2));
  }
- true is printed!
A Better equals()

- Take Object as parameter
- Check for non-null-ness of parameter
- Check that class type is correct
- Then do other checks
- For example in Person:
  ```
  public boolean equals (Object o) {
    if (o == null)
      return false;
    else if (o.getClass() != getClass())
      return false;
    else {
      Person p = (Person)o;
      return name.equals(p.getName()) &&
      idNum.equals(p.getIdNum());
    }
  }
  ```
- Similar improvements can be made to Student, Faculty
- Now bob.equals(bob2) returns false

Recall Interfaces

- Interfaces contain lists of method prototypes
- Example from Lecture #23:
  ```java
  public interface UMStudent {
    public void goToClass();
    public void study();
    public void add(int a, int b);
    public String getName();
  }
  ```
- Classes can be indicated as implementing interfaces
  ```java
  public class CSMajor implements UMStudent {
  }
  ```
- To satisfy Java compiler, CSMajor must provide implementations of goToClass(), study(), etc.
- Interfaces can be used as types, and thus to support polymorphism:
  ```java
  public void psychoAnalyze(UMStudent student) { ... }
  ```
- From last time: interfaces are similar to, but different from, abstract classes
  - Abstract classes can contain abstract, concrete methods
  - Classes can implement multiple interfaces, but inherit (directly) from only one class
Main Uses of Interfaces

- API for classes
- Polymorphism
- “Faking multiple inheritance”
- Specifying sets of symbolic constants

“Multiple Inheritance”? 

- Intuitively useful to be able to inherit from multiple classes (multiple inheritance)

- But Java does not allow this
Why Does Java Disallow Multiple Inheritance?

- Semantic difficulties!
- Consider StudentAthlete
  - Objects would get name field from Student
  - Objects would also get name field from Athlete
  - Duplicate fields: what to do?
- Some languages (e.g. C++) do allow multiple inheritance

Can We Achieve Some of Benefits of Multiple Inheritance in Java?

- Yes, using interfaces + inheritance
  - Idea: use inheritance for one of inherited classes, interfaces for others
  - Interfaces ensure that relevant methods are implemented
- Example
  ```java
  public class Person { … }
  public class Student extends Person { … }
  public interface Athlete {
    public String getSport ();
    public void setSport (String sport);
  }
  public class StudentAthlete extends Student implements Athlete { … }
  ```
  - Objects of type StudentAthlete "are" Students
  - They also can be wherever objects matching Athlete are required
Interfaces and Constants

- Interfaces can also contain public final static variables
- Sometimes interfaces are used to provide consistent definitions for constants throughout an application
- Example
  ```java
  public interface Months {
      public final static int JANUARY = 1;
      public final static int FEBRUARY = 2;
      public final static int MARCH = 3;
      ...
      public final static int DECEMBER = 12;
  }
  public class MonthDemo implements Months {
      public static void main( String[] args ) {
          System.out.println( "March is month number " + MARCH );
      }
  }
  
  Because MonthDemo implements Months, it has access to the constants
  ```

Interface Hierarchies

- Inheritance may also be used to build new interfaces from previous ones
- A subinterface inherits all method / constant declarations from its base interface
- A subinterface may also introduce new methods / constants
- E.g. public interface Level1<T> {
          boolean x();
          T y();
          void z();
      }

      We can define a new, bidirectional iterator interface using inheritance
      public interface Level2<T> extends Level1<T> {
          boolean a();
          T b();
      }