Inheritance and Interfaces

• what is inheritance?
• examples & Java API examples
• inheriting a method
• overriding a method
• polymorphism
• Object
  – toString
• interfaces
  – Ex: sorting and Comparable interface
Announcements

• Time to get started on PA3
• A few office hours changes this week. See piazza announcements.
Inheritance

- terminology: a subclass (or derived class) inherits from a superclass (or base class)
- derived class is a specialization of the base class
  - add or change functionality
  - reuse code and interface
  - can use derived objects in place of base objects.
- inheritance models IS-A or IS-A-KIND-OF relationship
- Some examples of this:
  - Dog IS-A Mammal
  - Manager IS-A Employee
  - Ford IS-A Car
Inheritance: what it isn’t

• review: inheritance models IS-A
• inheritance is not for HAS-A
  – Examples of HAS-A:
    • Car HAS Wheels
    • ArrayList HAS Elements
  – use containment for HAS-A
• a superclass is not a generic type
  – e.g., List vs. ListofInts vs. ListofStrings
  – Java generics does this: ArrayList<Integer>, ArrayList<String>
Some examples of inheritance

Calendar
  \[\text{GregorianCalendar}\]

Shape
  \[\text{Circle, Triangle, Rectangle}\]

Employee
  \[\text{Manager}\]

JComponent

CarComponent  CoinSimComponent
Inheriting a method

• From lab2:
  
  ```java
  GregorianCalendar date = ...;
  date.set(...);
  ```

• Gregorian calendar is a subclass of Calendar:
  
  ```java
  public class GregorianCalendar extends Calendar {
  
  set method is inherited from Calendar
  
  GregorianCalendar has no method definition for set
  ```
Overriding a method

• Making a subclass and
• **overriding** a method from the superclass

```java
public class CarComponent extends JComponent {
    // code to draw a car on the screen
}
```
Not method overriding

• method overloading:

```java
public class String {
    public String substring(int begin, int end) { ... }

    // return the substring that goes from the
    // specified index to the end of the string
    public String substring(int begin) { ... }
    . . .
}
```
Not method overriding

• Method signature different from the one defined in the superclass:

```java
public class CarComponent extends JComponent {
  . . .
  public void paintComponent(int length) {
    // code to draw a car on the screen
  }
}
```
Not method overriding

• Two unrelated classes with the same method name and params:

```java
// no inheritance – this paintComponent is unrelated
// to JComponent’s version
public class Foo{
    public void paintComponent(Graphics g) {
        . . .
    }
    . . .
}
```
Some characteristics of inheritance

- Can assign *up* the type hierarchy safely:

  ```java
  JComponent comp = new CarComponent(...);
  ```

  or

  ```java
  myFrame.add(new CarComponent(...));
  ```
Swing *using* CarComponent

- Java Swing framework code doesn’t know about `CarComponent`

- Java Swing code can later safely call:
  ```java
  component.paintComponent(g);
  ```

  *compile-time type* `JComponent`

- CarComponent’s `paintComponent` gets called *(run-time type)*
Polymorphism

• Varying what actual method is called at run-time via method overriding: polymorphism

• Overriding / polymorphism is type-safe

• All JComponent subclasses have to either inherit paintComponent or override it.
How is it type-safe?

```java
public class CarComponent extends JComponent {
    // overridden from JComponent:
    public void paintComponent(Graphics g) {...}

    // CarComponent-specific function:
    public Wheels getWheels() {...}
}
```

(Reminder: Foo defines paintComponent, it’s not a subclass of JComponent)

```java
myFrame.add(new Foo()); // 1
JComponent comp = new CarComponent(); // 2
comp.getWheels(); // 3
CarComponent carComp = // 4
    (CarComponent) new JComponent();
carComp.getWheels(); // 5
```
Object class

• Object is the highest class in the hierarchy
• Every other Java class is a subclass of Object
• (Might be a few levels down a hierarchy.)
• Means all objects have some methods in common:

```java
public class Object {
    public String toString() {...}
    public boolean equals(Object other) {
        {...}
    }
    ...
}
```
**toString method**

- Defined for all objects
- String “+” operator uses it automatically to convert your object type to a string:

```
System.out.println("My MazeCoord: " + mazeCoord);
```

- Calls `Object.toString` behind the scenes
- Default (**Object**) version prints weird stuff (hashcode)
- Convention: override `toString` to print out all the field names and values for debugging purposes
- Most Java classes override `toString` to do this.
- Ex: **Person** class
Example of defining `toString`

```java
public class Person {
    private String name;
    private int favoriteNumber;
    private Point geoCoord;
    public String toString() {
        return "Person[name=" + name
               + ",favoriteNumber=" + favoriteNumber
               + ",geocoord=" + geoCoord
               // calls Point toString
               + "]";
    }
}
```

Inheritance and Interfaces [Bono] 17
Interfaces

• `interface` and `implements` are Java keywords
• Like a superclass, but has no implementation of its own:
  – no instance variables
  – no method bodies
• Defines the headers for methods an implementing class must implement
• class that `implements` the interface…
  – may also have other methods
  – may implement multiple interfaces simultaneously
Ex: implementing an interface

• Part of Java library is `Comparable` interface:
  – implementing this interface means you can compare two objects of your type (less than, greater than)
  – . . . using a method called `compareTo`.
  – Some Java classes are `Comparable`, e.g., `String`, `GregorianCalendar`

• Example: make `Student` class comparable
Comparable interface

- A class is **Comparable** if it implements the `compareTo` method.

```java
public interface Comparable<Type> {
    int compareTo(Type other);
}
```
Comparable interface (cont.)

• Implementing comparable means clients can compare two objects of your type

• `String` implements `Comparable`

• `a.compareTo(b);`
  – returns $< 0$ if $a < b$
  – returns $> 0$ if $a > b$
  – returns $0$ if $a == b$

• What do we need to do to make our class comparable:
  – Declare that class implements `Comparable`
  – Implement `compareTo` method for our class
Implementing Comparable

class Student implements Comparable<Student> {
    private String firstName;
    private String lastName;
    private int score;
    ...
    public int compareTo(Student b) {
        int lastDiff = lastName.compareTo(b.lastName);
        if (lastDiff != 0) {
            return lastDiff;
        } else {               // last names are equal
            return firstName.compareTo(b.firstName);
        }
    }
}

Inheritance and Interfaces [Bono]
Sorting example

- Want to use `Arrays.sort`
- Sort is overloaded for `int[]`, `double[]`, etc.:
  ```java
  int[] myArr = ...;
  Arrays.sort(myArr);
  ```
- Uses `<` to compare two elements.
- But how to use sort on array of your own object types?
  ```java
  Student[] studArr = ...;
  Arrays.sort(studArr);
  ```
  - problem: `<` not defined for `Student`
  - What does it mean for one student to be less than another?
Sorting example (cont.)

• We can define what less-than means for Students
• But, we don't want to have to implement a sort routine ourselves.
  … And then reimplement for the next element-type we want to sort, etc.
• Solution: Sort has a version that works if our element-type implements the `Comparable` interface:
  ```java
class Arrays {
  . . .
  public static void sort(Comparable[] arr);
  ```
• What code do you need to write?
  1. Make Student class implement Comparable
     – part of that is to implement compareTo

  2. Now can use Java’s sort method on an array of Students:

    ```java
    Student[] studArr = ...;
    . . .
    Arrays.sort(studArr);
    ```

• Arrays.sort calls the compareTo method we defined
Code examples on-line

• In code directory for today’s lecture:
  • **Person** class (with **toString**) and tester program that shows the limits of when **toString** will automatically get invoked.
  • **compareEx** subdirectory:
    – **Student** class that implements **Comparable**
    – **Comparator** for two **Student** objects (part of readings: Special Topic 14.4)
    – Example prog that uses both of these to sort an array of **Student**’s two different ways.
Why extend a Java class or implement a Java interface?

• A common use of inheritance is to extend classes or implement interfaces defined by some library:
  – This is a way to plug in application specific code so other parts of the library can call our method without having to know anything about our exact class.

• Form of reusability. Today’s examples:
  – can reuse all the Swing GUI code with our own GUI app (Swing is an *application framework*)
  – can reuse the fast sort code to sort our own data