Inheritance and Interfaces

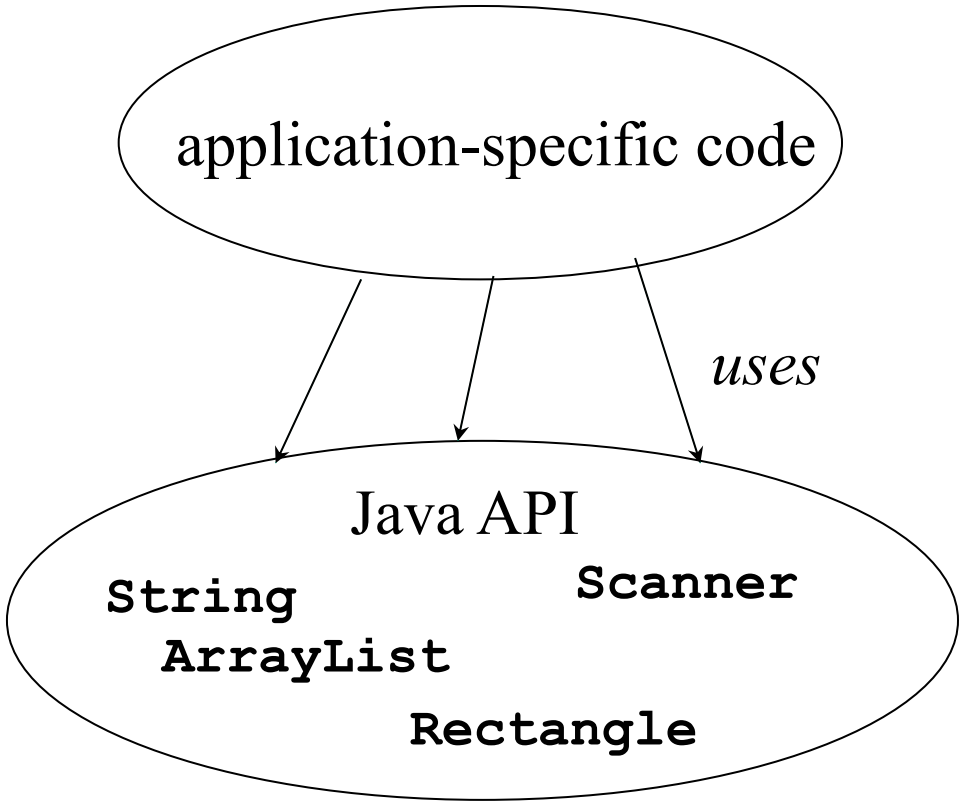
- what is inheritance?
- examples & Java API examples
- inheriting a method
- overriding a method
- polymorphism
- Object
  - toString
- interfaces
  - Ex: sorting and Comparable interface

Announcements

- If you haven't already: time to get started on PA 3: Step 1. Play a game!

So far: classes for code reuse

- One of the benefits of OOP is code reuse.



What is inheritance for?

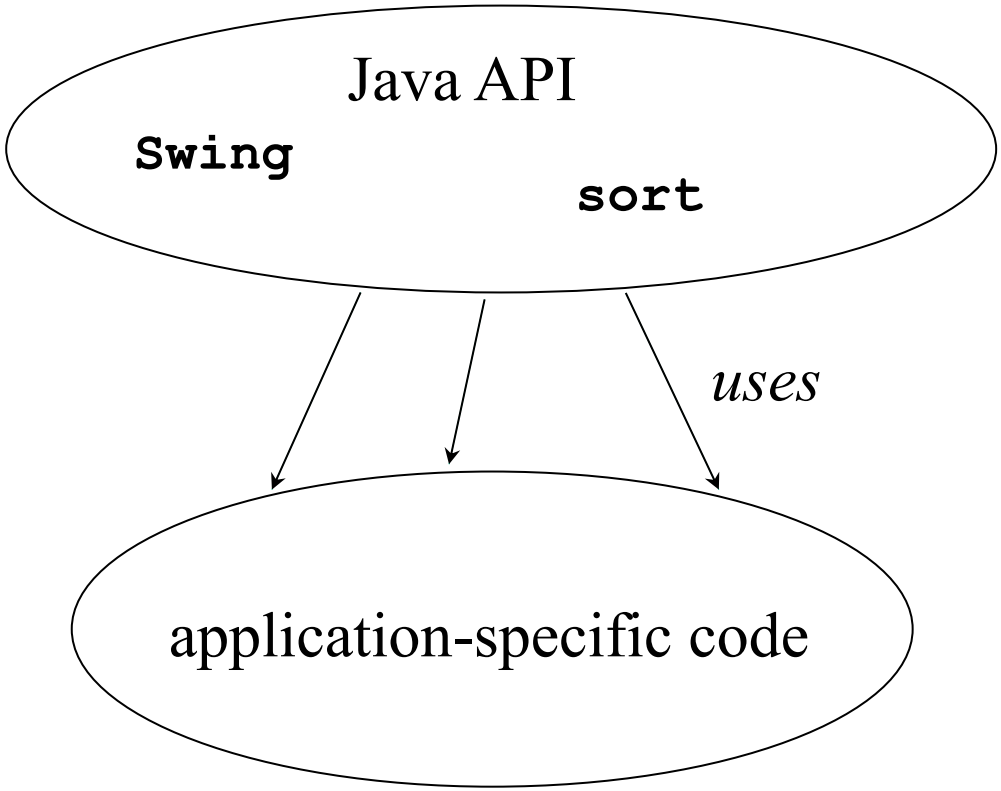
A more flexible form of code reuse

- from a library: can *customize* library code to use with our application (major example: GUI code)

- in an application: can take advantage of commonalities between different kinds of objects – may have different code inside them, but are used in similar ways in the application.

Customizing library code

- Customize library code to use with our application
- Library code calls our code



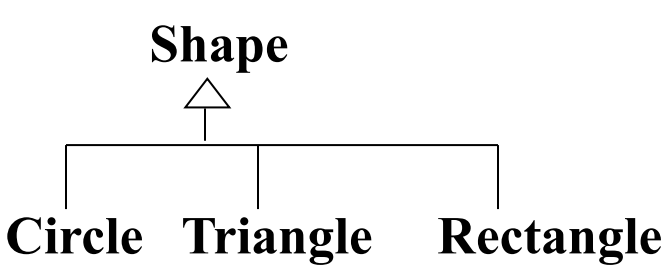
Inheritance

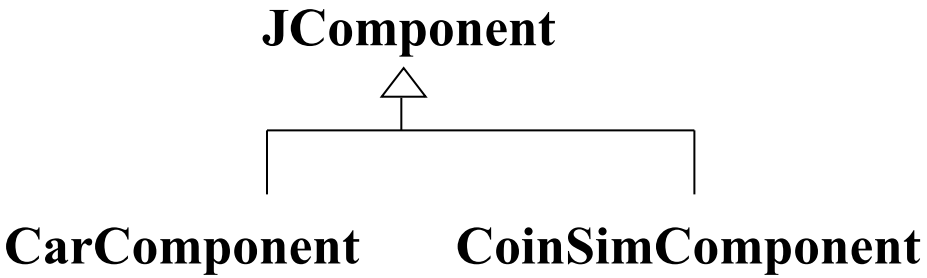
- terminology: a subclass (or *derived* class) inherits from a superclass (or *base* class)
- subclass class is a *specialization* of the superclass
  - add or change functionality
  - reuse code and interface
  - can use subclass objects in place of superclasss objects.
- inheritance models *IS-A* or *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 do this: ArrayList<Integer>, ArrayList<String>

Some examples of inheritance





Inheriting a method

- CarComponent class from textbook:
- CarComponent is a subclass of JComponent:

```
public class CarComponent
                        extends JComponent {
```

- **`getWidth()`** method is inherited from **`JComponent`**
- **`CarComponent`** has no method definition for **`getWidth()`**
- Example of calling the inherited method in client code:

```
CarComponent carComponent = …;
carComponent.getWidth(…);
```

Inheriting a method (cont.)

- Example where CarComponent itself calls the inherited method:

```
public class CarComponent extends JComponent {

   public void paintComponent(Graphics g) {
      . . .
      int x = this.getWidth() - 60;
      . . .
   }
}
```

Overriding a method

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

```
public class CarComponent extends JComponent {
    . . .
    public void paintComponent(Graphics g) {
       // code to draw a car on the screen
    }
}
```

Not method overriding (1)

- method **over*load*ing**:

```
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 (2)

- Method signature different from the one defined in the superclass (also overloading):

```
public class CarComponent extends JComponent {
   . . .
   public void paintComponent(int length) {
      // code to draw a car on the screen
   }
   public void paintComponent(Graphics g, int x) {
      // code to draw a car on the screen
   }
}
```

Not method overriding (3)

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

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

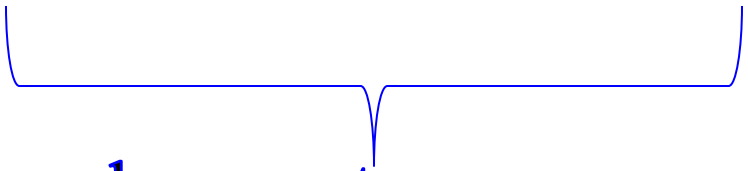
Some characteristics of inheritance

- Can assign *up* the type hierarchy safely:

```
JComponent comp = new CarComponent(…);
```

or

```
myFrame.add(new CarComponent(…));
```



Swing *using* CarComponent

- Java Swing framework code doesn't know about **`CarComponent`**

- Java Swing code can later safely call:

```
component.paintComponent(g);
```

compile-time type is **`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?

```
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`**)

```
myFrame.add(new Foo());                          // 1
JComponent comp = new CarComponent();            // 2
comp.paintComponent(g);                          // 3
Wheels w = comp.getWheels();                     // 4
CarComponent carComp = (CarComponent) comp;      // 5
carComp.getWheels();                             // 6
```

Asynchronous participation: Link to Inheritance poll

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:

```
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 account: " + bankAccount);
```

- 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`

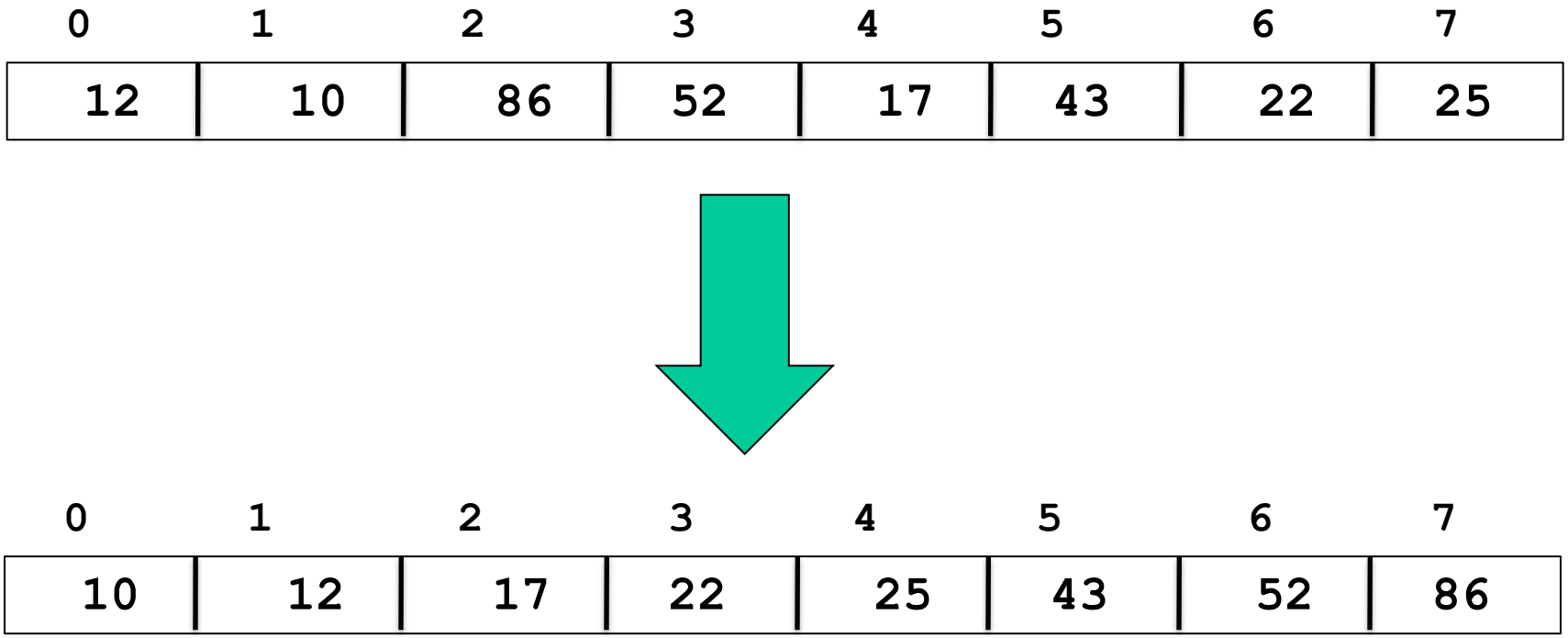
```
public class Person {
  private String name;
  private int favoriteNumber;
  private Point geoCoord;
  public String toString() {
    return "Person[name=" + name
       + ",favoriteNumber=" + favoriteNumber
       + ",geocoord=" + geoCoord
                          // calls Point toString
     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

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

Application of interfaces: Sorting in Java

The sorting problem:



Sorting example

- Java library provides **`Arrays.sort`** method
- Sort is overloaded for **`int[]`**, **`double[]`**, etc.:

```
int[] myArr = …;
Arrays.sort(myArr);
```

- Uses < to compare two elements.
- But how to use sort on array of your own object types?

```
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 function 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:

```
class Arrays {
  . . .
  public static void sort(Comparable[] arr);
```

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`**, **`LocalDate`** (from Lab 2)
- Example: make **`Student`** class comparable so we can sort arrays of students using Java sort method

Comparable interface

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

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

(Speical Topic 10.5 covers generic version of Comparable)

Comparable interface (cont.)

- Implementing comparable means clients can compare two objects of your type
- **`String`** implements **`Comparable`**:
- **`a.compareTo(b);`**
  - returns a value < 0 if a < b
  - returns a value > 0 if a > b
  - returns 0 if a = b
- What do we need to do to make our class comparable:
  - Declare that the 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);
    }
  }
}
```

Back to sorting-students problem

- 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:

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

- **`Arrays.sort`** calls the **`compareTo`** method we defined

Code examples on-line

- In Vocareum 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 overides **`toString`** and **`equals`**
  - **`Student`** class also 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.

Summary: 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 Java **`sort`** method to sort our own data
- Enables us to customize parts of the Java library for our application