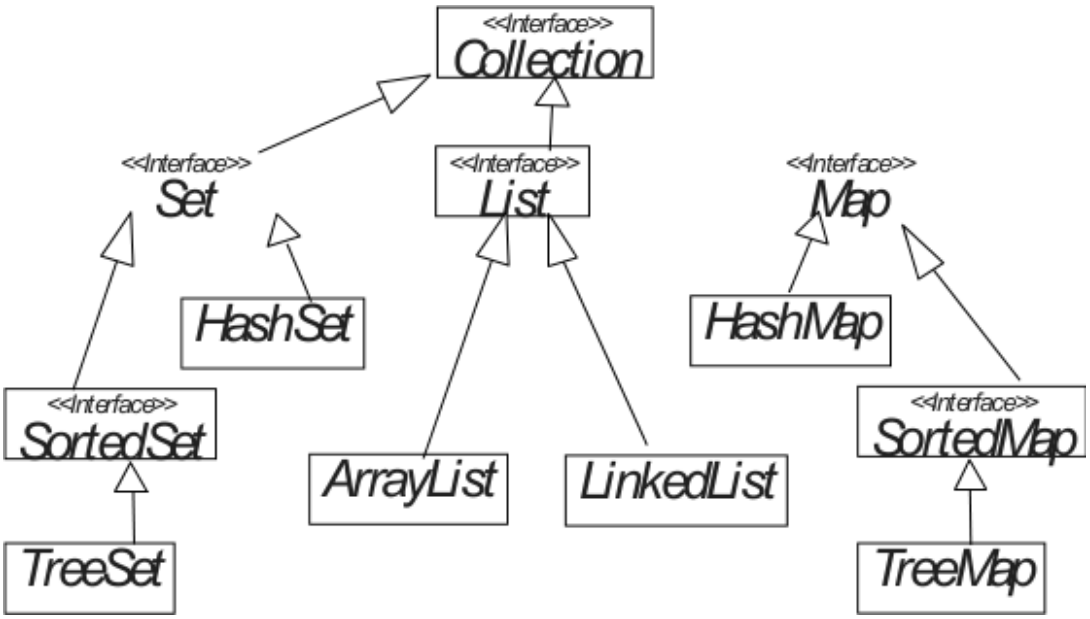
CSC2620 Object Oriented Design — Spring 2024
Unit 1 - Object Orientated Programming in Java

1.1 Review of Collections Framework

`http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`

- A **collection** is a data structure used to group related objects.
- A collection does not need previous knowledge about how big the collection will be (as opposed to an array, which needs to know at instantiation how large it will be, and it is then fixed at that size).
- List
- Map
- Set
- Queue
- ArrayList - implementation of List
- LinkedList - another implementation of List
- HashSet - implementation of a Set
- TreeSet - slower but provides a `SortedSet` (which means the keys must be `Comparable`)
- HashMap - implementation of a Map
- TreeMap - slower, but creates a `SortedMap` (which means the keys must be `Comparable`)



Sort collections using `Collections.sort(Collection<Comparable>)`.

1.2 ArrayLists and Collections Methods

- The package `java.util` provides the `ArrayList<T>` collection, which mimics the functionality of an array but can *dynamically* change its size depending on its contents.
- The `<T>` is a placeholder (used for **generic classes**, *not* primitive types). When an `ArrayList` is declared, replace the `T` with the type of object/data that will be stored in it. For instance, if you want an `ArrayList` of elements of type `Integer`, the declaration
would be `ArrayList< Integer > newIntArray = new ArrayList< >();`
- Common methods within the `ArrayList<T>` class:

Method	Description
`add`	Adds an element to the end of the list.
`clear`	Removes all elements from the list.
`contains`	Returns `true` if the list contains a specified element, `false` otherwise.
`get`	Returns the element at a specified index.
`indexOf`	Returns index of first occurrence of a specified element.
`remove`	Overloaded: Removes element at specified index, or the first instance of the specified element within the list.
`size`	Returns the number of elements stored in the list.
`trimToSize`	Trims the capacity of the list to the current number of elements.
`set`	Sets the element at ar particular index to a particular value.

Practice

1. Write a program that generates 1000000 random numbers between 1 and 10000.
2. Store those numbers in an `ArrayList`, a `TreeSet`, and a `Hashmap`.
   (a) Store all the numbers in an `ArrayList` normally.
   (b) Store all the numbers in a `TreeSet` normally. Compare the sizes of the `ArrayList` to the `TreeSet`...what do you notice? Print the contents (the first 100 elements should be sufficient, you won't be able to show the whole Collection).
   (c) Create a *hash table* of size 127 (why that size?) using the `HashMap`. How can you do this?
   (d) Print out the hash table by printing the contents of each "bucket" along with its hash value. Make this printing pretty.
3. Open the file `words.txt` and read them into a hash table, deciding on the best hash value you can think of. Use this hash table to create a tool that allows a user to input a word and prints "Yes, that word exists in the list of words!" or "No, that word does not exist in the list of words.".

1.3 Object Oriented Programming

- The Object Oriented programming paradigm aims to help overcome these problems by helping with the analysis and design tasks during the initial software development phase and by ensuring software is robust and maintainable
- **Abstraction** and **encapsulation** are the *fundamental principles* of Object Oriented Programming.
  - **Abstraction** allows the programmer to ignore details by concentrating on complex global ideas.
  - **Encapsulation** allows the programmer to focus on what something does instead of worrying about the complxities of how it does what it does.
- The two other primary tenets of Object Oriented Programming are **Inheritance** (using Generalization and Specialization) and **Polymorphism**, covered later in the semester.
- We can define general characteristics (Generalization) for all objects, and then create individual, specialized versions of those objects for specific purposes (Specialization). This is the essence of Object Oriented Programming.

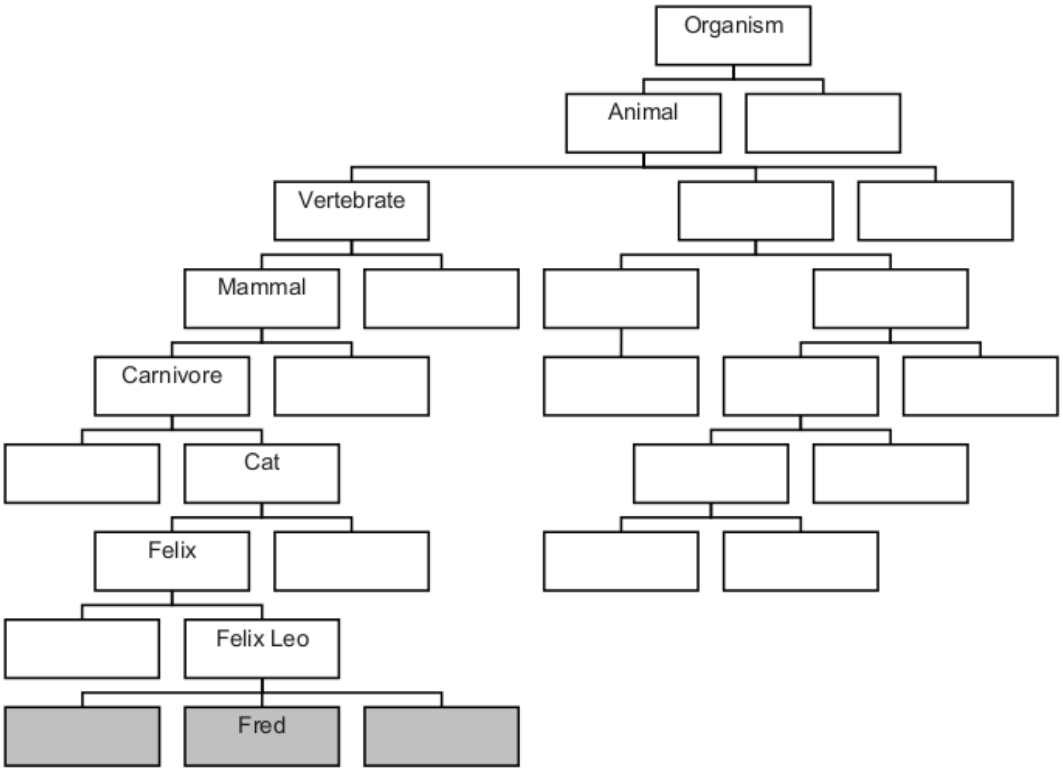
1.4 Review of Inheritance

- **Inheritance** is a form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities. It revolves around categorizing objects by "families" that "pass down" behaviors and attributes.
- Inheritance can save time during program development by basing new classes on existing proven and debugged high-quality software.
  - It increases the likelihood that a system will be implemented and maintained effectively.
- When creating a class, rather than declaring completely new members, you can designate that the new class should **inherit** the members of an existing class.
  - The existing class is the **superclass**.
  - The new class is the **subclass**.
- Each subclass can be a superclass of future subclasses. There are no limitations on the levels of the class heirarchy that is created by the inheritance structure.
- A subclass can add its own fields and methods, beyond those implemented by the superclass. These are generally attributes and behaviors that are local to the subclass but not shared with the superclass.
- A subclass is more specific than its superclass and represents a more specialized group of objects.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass. This is why inheritance is sometimes referred to as **specialization**.
- The **direct superclass** is the superclass from which the subclass explicitly inherits.
- An **indirect superclass** is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with class `Object` (in package `java.lang`). Every class in Java directly or indirectly extends (or "inherits from") `Object`.
- Java supports only **single inheritance**, in which each subclass is derived from exactly one direct superclass. Some other object-oriented languages, such as C++, allow a class to inherit from multiple superclasses.

1.5 Class Relationships

- We distinguish between the *is-a* relationship and the *has-a* relationship.
- *Has-a* represents composition. In a *has-a* relationship, an object contains as members references to other objects.
  - A `Automobile` *has a* `Color`.
  - A `JPanel` *has a* `GridLayout`.
  - A `Library` *has* `Songs`.
- *Is-a* represents inheritance. In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass.
  - A `Car` *is an* `Automobile`.
  - A `Square` *is a* `Rectangle`.
  - A `JButton` *is a* `Component`.
- Superclasses tend to be "more general" and subclasses "more specific."
- Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses.

- There are several more examples



- A superclass exists in a hierarchical relationship with its subclasses (also called an **inheritance hierarchy**). Take a look at this sample university community class hierarchy:

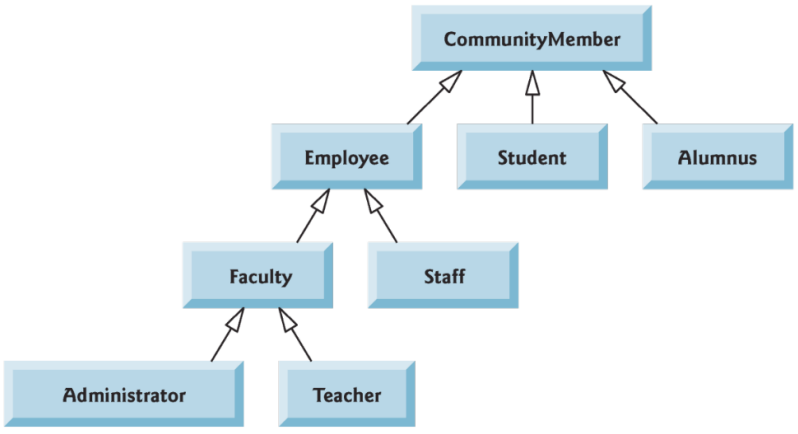


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

  - Each arrow in the hierarchy represents an *is-a* relationship.
  - Follow the arrows upward in the class hierarchy:
    - An `Employee` *is a* `CommunityMember`.
    - A `Teacher` *is a* `FacultyMember`.
  - `CommunityMember` is the direct superclass of `Employee`, `Student` and `Alumnus` and is an indirect superclass of all the other classes in the diagram.
  - Starting from the bottom, you can follow the arrows and apply the *is-a* relationship up to the topmost superclass.

- Similarly, a fairly typical class hierarchy is the `Shape` hierarchy:

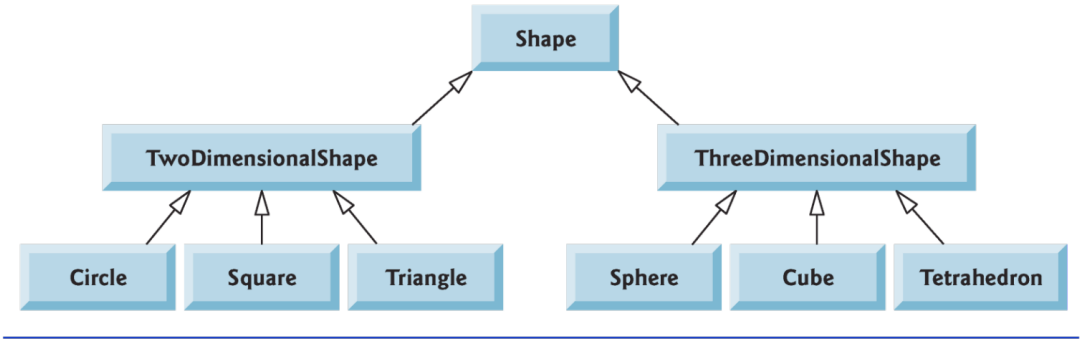


Fig. 9.3 | Inheritance hierarchy for Shapes.

- We can follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships.
  - A `Triangle` *is a* `TwoDimensionalShape` and is a `Shape`.
  - A `Sphere` is a `ThreeDimensionalShape` and is a `Shape`.
- Not every class relationship is an inheritance relationship. Given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`. However, an Employee *has a* `BirthDate` and a `TelephoneNumber`.
- A subclass can inherit methods that it does not need or should not have. Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method. The subclass can override (redefine) the superclass method with an appropriate implementation.
  - To accomplish this, simply re-implement the method within the subclass.
  - If the method has the same signature (number of arguments, type of arguments, order of arguments) as that of the superclass's method, an instance of the subclass will use the subclass's method, and vice versa.

1.6 The Protected Keyword And Method Relationships

- A class's `public` members are accessible wherever the program has a reference to an object of that class, *or one of its subclasses.*
- A class's `private` members are accessible only within the class itself.
- `protected` access is an intermediate level of access between `public` and `private`. A superclass's `protected` members can be accessed by members of that superclass, by members of its subclasses and by members of *other classes in the same package.*
- All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass.
- A superclass's `private` members are hidden in its subclasses. They can be accessed only through the `public` or `protected` methods inherited from the superclass.
- Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.
- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword `super` and a dot (.) separator:
  `super.getField()`.

1.7 Constructors of Subclasses

- Instantiating a subclass object begins a chain of constructor calls. The subclass constructor, before performing its own tasks, invokes its direct superclass's constructor.
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is always class `Object`'s constructor.
- The original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables the subclass object inherits.

1.8 Abstract Classes

- Sometimes it's useful to declare classes for which you never intend to create objects. They are to be used only as superclasses in inheritance hierarchies, so they are sometimes called **abstract superclasses**.
- They cannot be used to instantiate objects because they are incomplete.
- Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.
- Classes that can be used to instantiate objects are called **concrete classes**, the opposite of **abstract classes**. Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects. They specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.
- You make a class abstract by declaring it with keyword `abstract`.
- An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword `abstract` in its declaration, as in
  `public abstract void draw(int numVerts); // abstract method`
- Abstract methods *do not provide implementations.*
- A class that contains abstract methods *must be an abstract class* even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.
- Constructors and static methods cannot be declared abstract.
- You cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables.
  - These can hold references to objects of any concrete class derived from those abstract superclasses.
  - Programs typically use such variables to manipulate subclass objects polymorphically.
- You can use abstract superclass names to invoke static methods declared in those abstract superclasses.
- As an example, a class `DeviceDriver` can be used by an operating system. New drivers are created regularly, and implement `DeviceDriver`'s `read` and `write` methods, abstracted in the superclass.

1.9 Review of Polymorphism

- **Polymorphism** enables you to "program in the general" rather than "program in the specific" by allowing you to write programs that process objects that share the same superclass as if they're all objects of the superclass; this can simplify programming.
- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as x-y coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second, namely `move`.
- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of **polymorphism**. The same message sent to a variety of objects has "many forms" of results – hence the term *polymorphism*.
- With polymorphism, we can design and implement systems that are easily *extensible*. New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
- The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

1.10 Examples of Polymorphism

1.10.1 Shapes

- If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
- Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`. These operations can also be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
- Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable. At execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

1.10.2 Extensibility

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code.
- To add new objects to our video game:
  - Build a class that `extends SpaceObject` and provides its own `draw` method implementation.
  - When objects of that class appear in the `SpaceObject` collection, the screen manager code invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.
  - So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.

1.11 Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object. Invoking a method on a subclass object via a superclass reference invokes the subclass functionality. The type of the referenced object, not the type of the variable, determines which method is called.
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types. This is allowed because each subclass object is an object of its superclass. However, a superclass object cannot be treated as a subclass object, because a superclass object is not an object of any of its subclasses. The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type, a technique known as **downcasting**, that enables a program to invoke subclass methods that are not in the superclass.

```
public class polyTest {
  public static void main( String[] args ) {
    CommissionEmployee cE = new CommissionEmployee( "Sue",
      "Jones", "222-22-2222", 10000, 0.06 );
    BasePlusCommissionEmployee bE =
      new BasePlusCommissionEmployee( "Bob","Lewis",
      "333-33-3333", 5000, 0.04, 300 );

    // Calls CommissionEmployee's toString
    System.out.println( cE.toString() );

    // Calls BasePlusCommissionEmployee's toString()
    System.out.println( bE.toString() );

    CommissionEmployee cE2 = bE;
    // Calls BasePlusCommissionEmployee's toString()
    System.out.println( cE2.toString() );
```

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called. The compiler allows this "crossover" because an object of a subclass is an object of its superclass (but not vice versa).
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use. This process is called **dynamic binding**.

1.12 Summary of Allowed Assignments

- There are four possible ways to assign superclass and subclass references to variables of superclass and subclass types.

1. Assigning a superclass reference to a superclass variable is straightforward.

```
Animal aBird = new Animal();
```

2. Assigning a subclass reference to a subclass variable is straightforward.

```
Bird aBird = new Bird();
```

3. Assigning a subclass reference to a superclass variable is safe, because the subclass object is an object of its superclass.

```
Animal aBird = new Bird();
```

   - The superclass variable can be used to refer only to superclass members.
   - If this code refers to subclass-only members through the superclass variable, the compiler reports errors.

4. Attempting to assign a superclass reference to a subclass variable is a compilation error.

```
Bird aBird = new Animal();   //compilation error
```

   - To avoid this error, the superclass reference must be cast to a subclass type explicitly.

```
Bird aBird = (Bird) new Animal();
```

   - At execution time, if the object to which the reference refers is not a subclass object, an exception will occur.
   - Use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

```
Animal aBird = new Animal();
aBird instanceof Animal; // returns true
aBird instanceof Bird;   // returns false
aBird = new Bird();
aBird instanceof Animal; // returns true
aBird instanceof Bird;   // returns true
```

1.13 Interfaces in Java

- An **interface** is a class that contains no fields and only `abstract` methods.
- Interfaces offer a capability requiring that unrelated classes implement a set of common methods, standardizing the ways in which things such as people and systems can interact with one another. They define *what* operations are necessary for an object, but not *how* the operations are carried out.
- An interface declaration begins with the keyword `interface` and contains only constants and abstract methods.
  - All interface members must be `public`.
  - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly `public abstract` methods.
  - All fields are implicitly `public`, `static` and `final`.
- To use an interface, a concrete class must specify that it `implements` the interface and must declare each method in the interface with specified signature.
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
- Declare an `interface` in place of a `public abstract` class if the class has no implementation details inside.
- Interfaces are usually `public`.

1.14 Practice

1. First, we will work on understanding the idea of an *interface*.

   - Create an abstract class called `Automobile`, and two subclasses `Car` and `Truck`.
   - `Car` should have a field for `topSpeed`, and for `acceleration`.
   - `Truck` should have a field for `horsePower` (the general unit of power for the engine).
   - Automobile should `implement Comparable`.
     - Make sure to implement the *necessary* `compareTo` methods (do all classes need them?)
       - A `Truck` object's *speed* is determined by 50% of its `horsePower`.
       - A `Car` object's *speed* is determined by its `topSpeed`.
     - Compare Automobiles by their *speed*s.
       - **Hint:** Remember that every object inherits from the `Object` class...so if you are unsure what a method will be passed in as an argument, you can always make it of type `Object` and then check using `instanceof`. This is using a powerful aspect of the Java language's polymorphic behavior.
       - For example, if you want to write a `compareTo` method in the `Car` class, you can have it take in an object of type `Object` as an argument, and then check if that object is a `Car`, a `Truck`, or something else using `instanceof`.
   - Create a `toString()` method for each class.
   - Use the following `main`:

```
public static void main(String[] args) {
  Automobile[] a = new Automobile[4];
  a[0] = new Car(50, 10);
  a[1] = new Truck(95);
  a[2] = new Car(42, 15);
  a[3] = new Truck(92);
  Arrays.sort( a );

  for( Automobile aut : a ) {
     System.out.println(aut);
  }
}
```

2. Doing all of that checking using `instanceof` is ugly...let's use an *interface* and polymorphism to remove it.

   - Create an interface called `Driveable`
     - Each `Automobile` should `implement` it.
     - `Driveable` should have an abstract method `int getSpeed()`. Each `Driveable` object's speed should be calculated as described in the previous section.
   - *Remove* your `compareTo` methods from `Car` and `Truck` and instead write a `comapreTo` method in `Automobile` that compares the top speed of two `Automobile` objects.
   - Change your array of `Automobile`s to an array of `Driveable` objects. Sort them.
   - Write a method in the `main` class that takes an array of `Driveable` objects and prints each one's speed. Use it to test the sorting.