Recitation 4

June 14, 2017

Agenda

1. Announcements
2. Overriding methods review
3. Overloading methods
4. Polymorphism
5. Interfaces and Abstract Classes Overview
6. Exam 1 Q&A

Announcements

- Homework 1 regrades are due **TOMORROW**, 6/15.
- Homework 4 has been released and is due on Tuesday, 6/27 at 11:55 pm. This is a two-week homework that is longer than previous homeworks so tell them to start early!
- Exam 1 regrades are due next Tuesday, 6/20.

Overriding Methods Review

- Overriding a method means providing a new definition of a superclass method in a subclass. We've been doing this all along with toString and equals, which are defined in java.lang.Object, the highest superclass of all Java classes.
- Remember!
  - The default implementation of equals in the Object class is just ==, which is not good for comparing objects, so we must override it.
- The `@Override` annotation is *optional*.
  - It tells the compiler that you are intending to override this method and the compiler will issue an error if a method that has this tag isn't actually overriding a method. It will check the **method signature** to find a matching one in the superclass.

Overloading Methods

- Same method name but different signature, meaning different param types/order.
- They've seen this with constructor overloading.
- Ex: the `+` operator behaves differently for ints vs Strings.

Polymorphism

Subclasses of a class can share some similarity with its parent class as well as define unique behavior. I know people hate on the `Animal` examples with Polymorphism, but remember that while we may have heard the example a million times and we want something new and different, if it resonates with the students be sure to put them first and use what is connecting with them.

- Any class that can pass more than one is-a test is considered to be polymorphic.
  - Hint: all objects are polymorphic.
- We can only access an object through a reference variable. The type of the reference variable determines what methods can be invoked on it.
  - We can reference subclasses as their superclasses or as the interfaces they might implement
  - When referencing an object it is referred to by its compile-time (static) type, but acts like its runtime (dynamic) type when it is actually running.
  - When compiling, the compiler will check to make sure that what we are invoking works with the reference type (static type, compile time type).
  - When running, the JVM will actually invoke whatever the object's runtime (dynamic) type is through virtual method invocation.
  - dynamic method binding - the method that is actually invoked is "bound" to the reference's dynamic type.
- We can get around the compiler by casting our references to other types
  - Essentially telling the compiler to "trust us" .
  - Be careful, make sure what you are casting to is within the same hierarchy and going from more specific to more general.

Forms of polymorphism in Java:

- Subtype polymorphism-all objects are polymorphic because they extend Object.
  - Ex: method overriding - subclasses and superclasses may behave differently for the same method. This is **dynamic** method binding, the method is bound at **runtime**.
- Ad-hoc or "static" polymorphism-polymorphic methods behave differently when applied to different types.
  - Ex: method overloading - methods with the same name have different behavior. This is **static** method binding, the method is bound at **compile-time**.

Interfaces and Abstract Classes

Briefly cover these, they will learn more about them in class tomorrow.

Interfaces

- "Contracts" to spell out how different parts of a program interact without having to know exactly how each piece is implemented.
- Can contain only constants, method signatures, default methods, static methods, and nested types.
  - All methods are public and abstract by default unless the keyword `default` or `static` is included in the header.
- Abstract method signatures have no braces and are terminated by a semicolon.
- To implement an interface in a class use the keyword `implements`, to extend an interface in an interface use the keyword `extends`. **Like extends like.**
- A concrete class must implement/override all of the abstract methods in an interface. An abstract class can leave methods abstract and simply inherit them.

Abstract Classes

- More similar to a regular class, can have instance variables and private members.
- A subclass of an abstract class must implement all of the abstract methods in the parent class or be declared abstract itself.

Exam 1 Q&A

- If anyone has not collected their test, pass them back, making sure to check their buzzcards. Answer any questions the students have including explaining/coding the answer, but **don't project the answer key on the board.**