Objectives

- Inheritance
  - Final methods, fields
- Packages
- Wrapper Classes
- More on Inheritance
  - Abstract Classes
  - Interfaces

Review

- How do we verify that an object variable is a certain type?
- How do we specify an object variable has a different type (a derived type)?
- What is the syntax for Javadoc comments?
- How has developing in Eclipse been going?

Code Review

- Compare and contrast the following code snippets:

```
for (int i = 1; i <= string.length(); i++){
   newString += string.charAt(string.length() - i);
}
```

```
for( int i=string.length()-1; i >=0 ; i-- ) {
   newString += string.charAt(i);
}
```

FINAL KEYWORD

Preventing Inheritance

- Sometimes, you do not want a class to derive from one of your classes
- A class that cannot be extended is known as a `final` class
- To make a class final, simply add the keyword `final` in front of the class definition:

```
public final class Rooster extends Chicken {
      . . .
}
```

- Example of `final` class: `System`

Final methods

- Can make a method `final`
  - Any class derived from this class cannot override the `final` methods

```
class Chicken {
      . . .
      public final String getName() { . . . }
      . . .
}
```

- By default, **all** methods in a `final` class are `final` methods.

Why `final` methods and classes?

- **Efficiency**
  - Compiler can replace a `final` method call with an inline method
    - Does not have to worry about another form of this method that belongs to a derived class
  - JVM does not need to determine which method to call dynamically
- **Safety**
  - No alternate form of the method; straightforward which version of the method you called

Sept 28, 2009 Sprenkle - CS209 7

PACKAGES

Sept 28, 2009 Sprenkle - CS209 8

Packages

- Hierarchical structure of Java classes
  - Directories of directories

```
java
 ├─lang
 │  ├─Object
 │  └─String
 ├─net
 └─util
    └─Date
```

Fully qualified name: `java.util.Date`

- Use `import` to access packages

Sept 28, 2009 Sprenkle - CS209 9

Standard Practice

- To reduce chance of a conflict between names of classes, put classes in packages
- Use `package` keyword to say that a class belongs to a package:
  - `package java.util;`
  - *First* line in class file
- Typically, use a unique prefix, similar to domain names
  - `com.ibm`
  - `edu.wlu.cs.logic`

Sept 28, 2009 Sprenkle - CS209 10

Importing Packages

- Can import one class at a time or all the classes within a package
- Examples:

```
import java.util.Date;
import java.io.*;
```

Import entire package

  - * form may increase compile time
    - BUT, no effect on run-time performance

Sept 28, 2009 Sprenkle - CS209 11

WRAPPER CLASSES

Sept 28, 2009 Sprenkle - CS209 12

Wrapper Classes

- **Wrapper class** for each primitive type
- Sometimes need an instance of an `Object`
  - To use to store in `HashMaps` and other `Collections`
- Include functionality of parsing their respective data types

```
int x = 10;
Integer y = new Integer(10);
```

Wrapper Classes

- **Autoboxing** – automatically create a wrapper object

```
// implicitly 11 converted to
// new Integer(11);
Integer y = 11;
```

- **Autounboxing** – automatically extract a primitive type

```
Integer x = new Integer(11);
int y = x.intValue();
int z = x; // implicitly, x is x.intValue();
```

Convert right side for whatever is needed on the left

Effective Java: Unnecessary Autoboxing

- Can you find the inefficiency from object creation?

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++ ) {
     sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} `Long` instances

- How to fix?

`Autobox.java`

Effective Java: Unnecessary Autoboxing

- Can you find the inefficiency from object creation?

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++ ) {
     sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} `Long` instances

- How to fix?
- Lessons:
  - Prefer primitives to boxed primitives
  - Watch for unintentional autoboxing

`Autobox.java`

ABSTRACT CLASSES

Abstract Classes

- Some methods defined, others not defined
- Classes in which not all methods are implemented are ***abstract classes***
  - `public abstract class ZooAnimal`
- Blank methods are labeled as `abstract`
  - `public abstract void exercise();`

Abstract Classes

- An abstract class can***not*** be instantiated
  - i.e., can't create an object of that class
  - But can have a constructor!
- Child class of an abstract class can only be instantiated if it overrides and implements **each abstract method** of its parent class
  - If subclass does not override all abstract methods, it is **also abstract**

Abstract Classes

- `static`, `private`, and `final` methods cannot be `abstract`
  - These cannot be overridden by a child class
- `final` class cannot contain abstract methods Why?
- A class can be abstract even if it has no abstract methods
  - Use when implementation is incomplete and is meant to serve as a parent class for subclass(es) that complete the implementation
- Can have array of objects of abstract class
  - Does dynamic dispatch for methods

Examples of abstract classes

- Example 1:
  - `java.net.Socket`
  - `java.net.SSLSocket` (abstract)
- Example 2:
  - `java.util.Calendar` (abstract)
  - `java.util.GregorianCalendar`

Summary: Defining Abstract Classes

- Define a class as `abstract` when have *partial implementation*

Better Organization of Game Classes

- `GamePiece` should be abstract
  - No default image associated with it
  - move method is abstract
- `Human` class should implement move method
  - From `GamePiece` class

INTERFACES

Interfaces

- Like abstract classes with **all** abstract methods
  - A set of requirements for classes to conform to
- Pure specification, no implementation
- Classes can `implement` one *or more* interfaces

Example of an Interface

- We can call `Arrays.sort()` on an array
- `Arrays.sort()` sorts arrays of any object class that implements the `Comparable` interface
- Classes that implement `Comparable` must provide a way to decide if one object is less than, greater than, or equal to another object

`java.lang.Comparable`

```
public interface Comparable {
     int compareTo(Object other);
}
```

- Any object that is `Comparable` must have a method named `compareTo`()
- Returns:
  - < 0 for less than
  - 0 for equals
  - > 0 for greater than
- Similar to Python's `__cmp__` method

Implementing an Interface

- In the class definition, specify that the class will `implement` the specific interface

```
public class Chicken implements Comparable
```

- Provide a definition for all methods specified in interface

How to determine Chicken order?

- What if made the Chicken class `Comparable`?

`Comparable` Chickens

One way: order by height

```
public class Chicken implements Comparable {
  . . .
  public int compareTo(Object otherObject) {
     Chicken other = (Chicken)otherObject;
     if (height < other.getHeight() )
          return -1;
     if (height > other.getHeight())
          return 1;
     return 0;
  }
}
```

What if otherObject is not a `Chicken`?

Update `Chicken.java`

`Comparable` Interface API

- Specifies what the `compareTo()` method should do:
  - Return a –1 if the first object is less than the second object (passed as a parameter)
  - Return a 1 if the second object (passed as a parameter) is less than the first object
  - Return a 0 if the two objects are equal
- Says what Java library classes implement `Comparable`

Interfaces

- Contain only object (*not class*) methods
- All methods are `public`
  - Implied if not explicit
  - Error to have protected or private (Why?)
- Fields are constants that are `static` and `final`
- A class can implement multiple interfaces
  - Separated by commas in definition

Testing for Interfaces

- Use the `instanceof` operator to see if an object implements an interface
  - e.g., to determine if an object can be compared to another object using the `Comparable` interface

```
if (obj instanceof Comparable) {
      // runs if whatever class obj is an instance of
      // implements the Comparable interface
}
else {
      // runs if it does not implement the interface
}
```

Interface Object Variables

- Can use an object variable to refer to an object of any class that implements an interface
- Using this object variable, can *only* access the interface's methods
- For example…

```
Object obj;
…
if (obj instanceof Comparable) {
      Comparable comp = (Comparable) obj;
      boolean res = comp.compareTo(obj2);
}
```

Interface Definitions

```
public interface Comparable {
      int compareTo(Object other);
}
```

- Do not *need* to specify methods as `public`
  - Interface methods are `public` by default

Interface Definitions and Inheritance

- Can extend interfaces
  - Allows a chain of interfaces that go from general to more specific
- For example, define an interface for an object that is capable of moving:

```
public interface Movable {
      void move(double x, double y);
}
```

Interface Definitions and Inheritance

- A powered vehicle is also `Movable`
  - Must also have a `milesPerGallon()` method, which will return its gas mileage

```
public interface Powered extends Movable {
      double milesPerGallon();
}
```

Constants in an Interface

- If a variable is specified in an interface, it is automatically a constant
  - `public static final variable`

```
public interface Powered extends Movable {
      double milesPerGallon();
      double SPEED_LIMIT = 95;
}
```

- An object that implements `Powered` interface has a constant `SPEED_LIMIT` defined

Interface Definitions and Inheritance

- `Powered` interface extends `Movable` interface
- An object that implements `Powered` interface must satisfy all requirements of that interface as well as the parent interface.
  - A `Powered` object must have a `milesPerGallon()` and `move()` method

Multiple Interfaces

- A class can implement multiple interfaces
  - Must fulfill the requirements of each interface
- But NOT possible with inheritance
  - A class can only extend (or inherit from) **one** class

```
public final class String implements
      Serializable, Comparable, CharSequence { …
```

Common Uses of Interfaces

- Define constants for multiple classes/ package
  - Something like global constants
  - However, not great design practice
- Marker Interface
  - Interface that is empty
  - Use to identify an object that has a certain property
    - E.g., `Cloneable`

Using an Interface or Abstract Class

Interfaces

- ✓ *Any* class can use
  - ✓ Can implement multiple interfaces
- No implementation
- – Implementing methods multiple times
- – Adding a method to interface will break classes that implement

Abstract Classes

- Contain partial implementation
- – Can't extend/subclass multiple classes
- ✓ Add non-abstract methods without breaking subclasses

One Option: Use Both!

- Define interface, e.g., `MyInterface`
- Define abstract class, e.g., `AbstractMyInterface`
  - Implements interface
  - Provides implementation for some methods

Abstract Classes and Interfaces

- Important structures in Java
- Will return to/apply these ideas throughout the course

Due Friday: Assignment 6

- Abstract classes practice
  - Make `GameObject` an `abstract` class
    - Define `move` as an abstract method
- Update Birthday's `equals` method
- Packages
  - Organize `MediaItem` classes into a package
- Interfaces practice
  - `MediaItem` and subclasses implement `Comparable` interface