Objectives

- Javadocs
- Inheritance
  - Final methods, fields
- Abstract Classes
- Interfaces

JAVADOCS

"Documentation is a love letter that you write to your future self." – Damian Conway

Javadocs

- Special comments, which are used to generate HTML documentation
- Syntax:

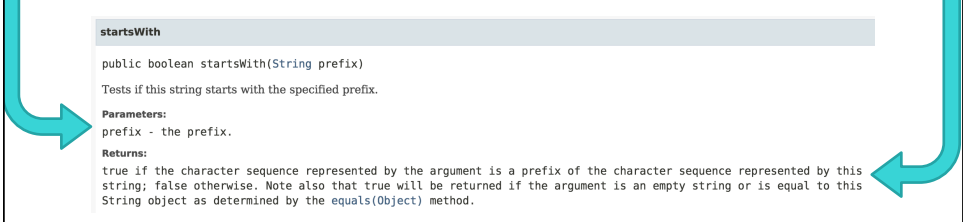
```
/**
 * Comment
 */
```

- Put before a class, a method, or a field to describe the respective class/method/field

Javadoc

- Can contain HTML syntax in description
- Example block tags to describe your code

```
@param <paramname> <description>
@return <description>   (include special cases)
```



startsWith

`public boolean startsWith(String prefix)`

Tests if this string starts with the specified prefix.

Parameters:
`prefix` - the prefix.

Returns:
`true` if the character sequence represented by the argument is a prefix of the character sequence represented by this string; `false` otherwise. Note also that `true` will be returned if the argument is an empty string or is equal to this `String` object as determined by the `equals(Object)` method.

Examples

```
/**
 * A simple Java class that models a Chicken.  The
 * state of the chicken is its name, height, and weight
 *
 * @author Sara Sprenkle
 */
```

Tags always go last in Javadoc comment

```
/**
 * @return the height of the chicken, in centimeters
 */
```

```
/**
 * @param n the String representing the name of the
chicken
 */
```

Expect these types of comments on all methods from now on

Generating Javadocs

- From command-line:

```
javadoc [options] [packagenames]
[sourcefiles] [@files]
```

- Generates HTML files
  - Example: Game's Javadocs

Review

- How do we make a class inherit from a parent class?
- How does a class refer to its parent class?
- What does a class inherit from its parent class?
  - What is *not* inherited?
- What are the access modifiers, ordered from least restrictive to most restrictive?
- How can we verify that an object variable is a certain type?
- How can we specify that an object variable has a different type (a derived type)?
- How does Java decide which method to call on an object?
  - Example: `chicken[1].feed();`

Review

- Designing classes: When should you make a variable/field
  - Local vs instance vs static
  - Private vs protected vs public
- Inheritance in game code
  - Java docs
- Importance of vocabulary

Summary of Inheritance

- Remove repetitive code by modeling the "is-a" hierarchy
  - Move "common denominator" code up the inheritance chain
- Don't use inheritance unless *all* inherited methods make sense
- Use polymorphism

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 would we want to use `final`?
What are possible benefits to us, the compiler, JVM,…?

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

ABSTRACT CLASSES

Abstract Classes

- Some methods defined, others not defined
  - Partial implementation
- 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(Environment env);`

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 **every abstract method** of parent class
  - If child class does not override *all* abstract methods, it is **also abstract**

Abstract Classes

- `static`, `private`, and `final` methods cannot be `abstract`
  - B/c 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 class(es) that complete the implementation
- Can have array of objects of abstract class
  - JVM will do dynamic dispatch for methods

Summary: Defining Abstract Classes

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

INTERFACES

Interfaces

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

Example of an Interface

- `Arrays.sort(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 (inherits) `Comparable` must have a method named `compareTo()`
- Returns:
  - Return a negative integer if the this object is less than the object passed as a parameter
  - Return a positive integer if the this object is greater than the object passed as a parameter
  - Return a 0 if the two objects are equal

Javadoc API of Comparable says which classes implement

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?

`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;
      // simpler: return height-other.getHeight()
  }
}
```

What if `otherObject` is not a `Chicken`?

We have a better way to handle soon…

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

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

- But NOT possible with inheritance
  - A class can only extend (or inherit from) **one** class

Benefits of Interfaces

- Abstraction
  - Separate the interface from the implementation
- Allow easier type substitution
  - We'll see this with Collections
- Can implement multiple interfaces

Interface Summary

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

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
  - Make code easier to change
- Will return to/apply these ideas throughout the course
- Concepts are used in many languages besides Java

TODO

- Assignment 5: due Wednesday