

Week 4 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Creating your own exceptions
- Throwing exceptions
- Exceptions and inheritance

Questions?

Project 1

Hoare's consequence rule

- Hoare's consequence rule says that a method can override a parent method as long as:
 1. Its parameters are broader (or the same)
 2. Its return value is narrower (or the same)
 - Think of exceptions thrown as things that can be "returned" too
- In other words, it will take even more kinds of input but will give back fewer kinds of output

Exceptions example

```
public class Vehicle {  
    public Trip ride(String destination) throws  
        CrashException, NauseaException {  
        return new Trip(destination);  
    }  
}
```

Narrower



```
public class Helicopter extends Vehicle {  
    @Override  
    public HelicopterTrip ride(String destination) throws  
        HelicopterCrashException {  
        return new HelicopterTrip(destination);  
    }  
}
```

Review

Java basics

- Primitive types: **byte, char, short, int, long, float, double, boolean**
- Operations: **+, -, *, /, %**, and shortcut versions
- Case sensitivity
- White space doesn't (usually) matter
- Three kinds of comments
- Arrays

Control structures

- Selection
 - `if`
 - `switch`
- Loops
 - `while`
 - `do-while`
 - `for`
 - Enhanced `for`
- `break` and `continue`: don't use them

Enhanced for loops

- Used to iterate over the contents of an array (or other collection of data)
- Similar to **for** loops in Python
- The **type** must match the elements of the array (or other collection)
- Syntax:

```
for (type value : array) {  
    // Statements  
    // Braces not needed for single statement  
}
```

Objects and methods

- Static methods do work but are not connected to objects
- Reference types are arrows to objects
 - More than one arrow can point at a single object
- Objects contain
 - Members
 - Methods
- Objects should be compared with the **equals()** method instead of **==**

Classes

- In addition to holding static methods, classes are template for objects
- Members (data) are usually private
- Methods (actions) are usually public
- Special kinds of methods:
 - Constructors specify how an object should be initialized
 - Accessors (getters) specify how an object can give back information
 - Mutators (setters) specify how an object changes data inside itself
- Static variables live in the class, not in an object (and shouldn't be used)
 - Unless they are constant (**final**)

Enums

- An enum is a special kind of class that has pre-defined constant objects
- These objects are intended to represent a fixed collection of named things:

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY  
}
```

- Individual days can be referenced like static variables: **Day.MONDAY** or **Day.FRIDAY**
- Since enum values are constants, it's convention to name them in ALL CAPS
- In addition to **int**, **char**, and **String** values, enums can be used for cases in **switch** statements

Packages

- To organize classes, they are often inside of packages
- This approach allows to tell the difference between two different classes with the same name that are in different libraries
- Packages correspond to folders with the same names
- Most packages are inside of other packages
- The default package (no package) should not be used for professional programming
- To use classes from other packages, import them:
 - `import java.util.Scanner;` or
 - `import java.util.*;`

Interfaces

Interface basics

- An **interface** is a set of methods which a class must have
- **Implementing** an interface means making a promise to define each of the listed methods
- It can do what it wants inside the body of each method, but it **must** have them to compile
- A class can implement as many interfaces as it wants

Interface definition

- An interface looks a lot like a class, but all its methods are generally empty
- Interfaces have no members except for (**static final**) constants

```
public interface Guitarist {  
    void strumChord(Chord chord);  
    void playMelody(Melody notes);  
}
```

Interfaces

- Many interfaces only have a single method
- Consider the following example:

```
public interface NoiseMaker {  
    String makeNoise ();  
}
```

- To implement this interface, a class must:
 - State that it implements the interface
 - Have a public, non-static method called **makeNoise ()** that takes no parameters and returns a **String**

Example classes

- Here are classes that implement **NoiseMaker**:

```
public class Pig implements NoiseMaker {
    public String makeNoise() {
        return "Grunt!";
    }
}
public class Explosion implements NoiseMaker {
    public String makeNoise() {
        return "BOOM!";
    }
}
public class Wind implements NoiseMaker {
    public String makeNoise() {
        return "Woosh!";
    }
}
```

Default methods

- As of Java 8, interfaces can also have default methods
- The interface expects you to implement these methods, but if you don't, a default implementation is provided

```
public interface Punchable {  
    default boolean wantsPunch() { // Default  
        return false;  
    }  
    void getPunched(Punch punch); // Abstract  
}
```

Interfaces can extend other interfaces

- Like classes, you can use inheritance to extend an interface
- When you do so, the child interface gets all of the required methods from the parent interface
- It can also reference the constants and static methods within the parent interface
- Consider the following interface:

```
public interface Defender {  
    boolean blockWithShield(Attack attack);  
}
```

Child interface

- We can make a child interface from **Defender** using the **extends** keyword

```
public interface NinjaDefender extends Defender {  
    boolean parryWithKatana(Attack attack);  
}
```

- This interface contains the **blockWithShield()** abstract method as well as the **parryWithKatana()** abstract method
- A class that implements this interface must have both

Inheritance

Inheritance

- The idea of inheritance is to take one class and generate a child class
- This child class has everything that the parent class has (members and methods)
- But you can also add more functionality to the child
- The child can be considered to be a **specialized** version of the parent

Subclass relationship

- Java respects the subclass relationship
- If you have a **Vehicle** reference, you can store a **Car** object in that reference
- A subclass (in this case a **Car**) is a more specific version of the superclass (**Vehicle**)
- For this reason, you can use a **Car** anywhere you can use a **Vehicle**
- You **cannot** use a **Vehicle** anywhere you would use a **Car**

Extending a superclass

- We use the **extends** keyword to create a subclass from a superclass

```
public class Car extends Vehicle {  
    private String model;  
    public Car(String s) { model = s; }  
  
    public String getModel() { return model; }  
  
    public void startEngine() {  
        System.out.println("Vroooooom!");  
    }  
}
```

- A **Car** can do everything that a **Vehicle** can, plus more

Subclass example

- As long as **Car** is a subclass of **Vehicle**, we can store a **Car** in a **Vehicle** reference

```
Vehicle v = new Car("Lancer Evolution"); // okay
```

- Even in an array is fine

```
Vehicle[] vehicles = new Vehicle[100];  
for( int i = 0; i < vehicles.length; i++ )  
    vehicles[i] = new RocketShip(); // cool
```

- Storing a **Vehicle** into a **Car** doesn't work

```
Car c = new Vehicle(); // gives error
```

Constructors

- A child class has to create a version of the parent class "inside" itself
- Consequently, the first line of a child class constructor is reserved for a call to the parent constructor
- If the parent has a default constructor (with no arguments), no call is necessary
- Otherwise, a call to the parent constructor **must** be made by using the keyword **super**, followed by parentheses and the arguments passed to the parent constructor

FoieGras class

- The **FoieGras** class extends **Food** and consequently must call the **Food** constructor as the first thing in its constructor
- The **FoieGras** constructor can be completely different from the **Food** constructor as long as it calls the **Food** constructor correctly

```
public class FoieGras extends Food {  
    private int grams;  
  
    public FoieGras(int grams) {  
        super("Foie Gras", 462*grams/100);  
        this.grams = grams;  
    }  
}
```

protected keyword

- In addition to public and private modifiers, the **protected** keyword is meaningful in the context of inheritance
 - Methods and members that are **public** can be accessed by any code
 - Methods and members that are **private** can only be accessed by methods from the same class
 - Methods and members that are **protected** can be accessed by code in the same package and by methods of any classes that inherit from the class
- Hard-core OOP people dislike the **protected** keyword since it allows child classes to fiddle with stuff that they probably shouldn't

Adding to existing classes is nice...

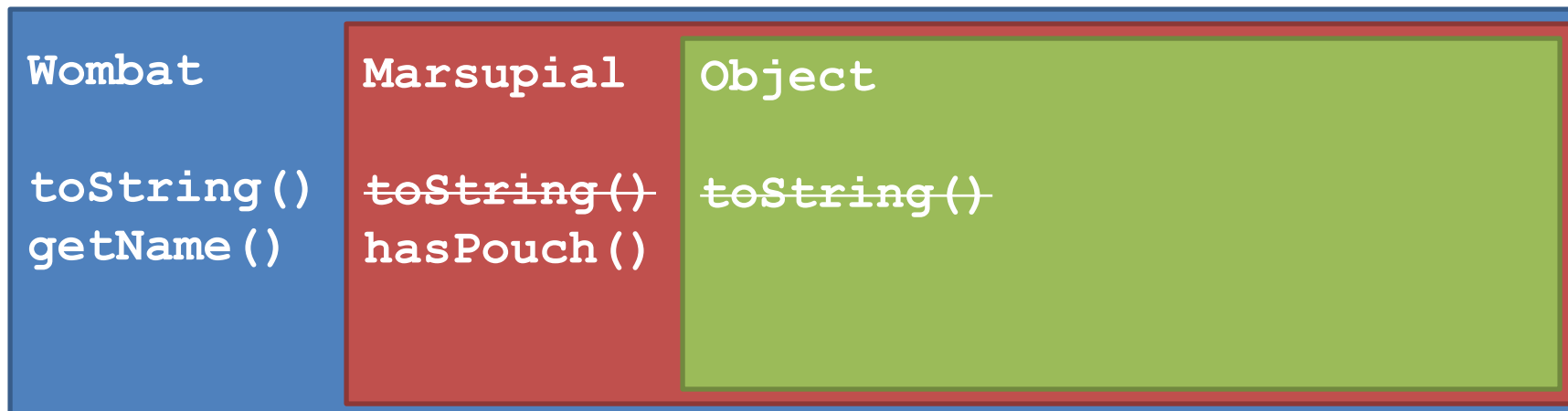
- Sometimes you want to do more than add
- You want to change a method to do something different
- You can write a method in a child class that has the same name as a method in a parent class
- The child version of the method will always get called
- This is called **overriding** a method

Dynamic binding

- All normal Java methods use dynamic binding
- This means that the most up-to-date version of a method is always called
 - It also means that the method called by a reference is often not known until run-time
- Consider a class **Wombat** which extends **Marsupial** which extends **Object**
- Let's say that **Wombat**, **Marsupial**, and **Object** all implement the **toString()** method

How to think about inheritance

- Every object has a copy of its parent object inside (which has its parent inside, and so on)
- All methods from the class and parents are available, but the outermost methods are always chosen
 - If a class overrides its parent's method, you always get the overridden method



The `final` keyword

- As you know, the **`final`** keyword is used to mark both member variables and local variables as constant
- **`final`** can be applied to methods and classes as well
- A **`final`** method cannot be overridden by a child class
- A **`final`** class cannot be extended at all
- **`String`** is an example of a **`final`** class
 - You can't extend **`String`** to make your own special kind of **`String`**!
 - We want **`String`** behavior to be totally consistent

Abstract methods

- All methods in interfaces are, by default, abstract
- An abstract method is only the signature of a method, not its definition
- Abstract methods end with a semicolon instead of a body defining what they do
- Any class that wants to implement the interface must complete all its abstract methods
- You can put abstract methods in classes, but
 - The method must be marked with the **abstract** keyword
 - The class must be abstract too

Abstract classes

- An **abstract class** is one that can't be instantiated
- It's intended to be the basis for inherited classes
- It's kind of like an interface in that it can contain abstract methods
 - But you can put regular methods in an abstract class
 - And member variables!
- An abstract class gives you a framework but not all of the implementation

Abstract class example

- The **Polygon** abstract class makes a foundation for polygons:

```
public abstract class Polygon {
    private final int sides;
    public Polygon(int sides) {
        this.sides = sides;
    }
    public final int getSides() {
        return sides;
    }
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

instanceof keyword

- Sometimes it's useful to know the true type of an object
- You can use the **instanceof** keyword to see if the type of an object inherits from a particular class
- Syntax (produces a **boolean**):
 - **object instanceof Class**
- An **instanceof** is almost always in an if statement:

```
Object object = getRandomObject();  
if(object instanceof Hurricane)  
    System.out.println("You can call me slurricane.");
```

More on instanceof

- **instanceof** doesn't tell you if an object is a particular class
- Instead, it tells you if it is that class or inherits from it
- Consider an object of type **Whiskey**, which inherits from **Alcohol**, which inherits from **Beverage** (which inherits from **Object**)

```
Object object = new Whiskey();
if(object instanceof Whiskey)           // true
    System.out.println("Whiskey!");
if(object instanceof Alcohol)           // true
    System.out.println("Alcohol!");
if(object instanceof Beverage)          // true
    System.out.println("Beverage!");
if(object instanceof Object)            // true
    System.out.println("Object!");
if(object instanceof String)            // false
    System.out.println("String?");
```

getClass () method

- For situations where you need to know if the type of an object matches exactly, you can use its **getClass ()** method
- This returns a **Class** object, which you can compare using **==** to the name of a type followed by **.class**

```
Object object = new Whiskey ();
if (object.getClass () == Whiskey.class) // true
    System.out.println ("Whiskey!");
if (object.getClass () == Alcohol.class) // false
    System.out.println ("Alcohol!");
if (object.getClass () == Beverage.class) // false
    System.out.println ("Beverage!");
if (object.getClass () == Object.class) // false
    System.out.println ("Object!");
```


Exceptions

Exceptions

- Instead of checking every method, Java has a general way of handling errors (and other exceptional situations)
- The name for this system is **exception handling**
- When an error happens, code will **throw** an exception
 - Throwing an exception usually means something went wrong
- A special block of code **catches** the exception
- When you catch an exception, you can
 - Deal with the problem and move on
 - Throw the same (or a new) exception and make someone else deal with it

Catching an exception

- The `risky()` method has a chance of destroying the world
- If the world is destroyed, execution will jump into the `catch` block

```
try {  
    System.out.println("About to do something risky!");  
    risky();  
    System.out.println("That was worth it!");  
}  
catch (WorldDestroyedException e) {  
    System.out.println("Whoops. We destroyed the world.");  
}
```

Multiple catch statements

- If a some code can cause many different exceptions, you can use multiple catches to handle them
- When a problem happens, execution will jump to the first catch that matches

```
try {
    useNumber(100 / divisor);
    getHoney();
    stayUpAllNight();
}
catch(ArithmeticException e) {
    System.out.println("We divided by zero!");
}
catch(BeeStingException e) {
    if(allergic)
        System.out.println("We're dying!");
    else
        System.out.println("Youch!");
}
catch(ExhaustedException e) {
    System.out.println("*YAWN*");
}
```

A **finally** block

- If an exception is thrown, the remaining code inside a **try** won't be executed
- If an exception isn't thrown, none of the **catch** blocks will be executed
- If you want code that is executed no matter what, it can be put in a **finally** block after all the **catch** blocks
- **finally** blocks are often used to do clean-up so we're sure it gets done
 - Things like closing files or network connections

The `throws` keyword

- If a method doesn't want to catch a (checked) exception, it must be marked as throwing that exception with the **`throws`** keyword

```
void pet (Goat goat) throws GoatBiteException {  
    goat.touch (); // can throw GoatBiteException  
}
```

- This **`pet ()`** method doesn't handle a **`GoatBiteException`** and thus must use the **`throws`** keyword to warn other code that it could throw a **`GoatBiteException`**

Creating an exception class

- Exceptions are classes like any other in Java
- They can have members, methods, and constructors
- All you need to do is make a class that extends **Exception**, the base class for all exceptions

```
public class SimpleException extends Exception {  
}
```

- That's it.
- Although it makes them long, it's good style to put the word **Exception** at the end of any exception class name

throw keyword

- The **throw** keyword is used to start the exception handling process
- You simply type **throw** and then the exception object that you want to throw
- Most of the time, you'll create a new exception object on the spot
 - Why would you have one lying around?

```
throw new CardiacArrestException ();
```

- Don't confuse it with the **throws** keyword!

Exception throwing example

- Here's a method that finds the integer square root of an integer

```
public static int squareRoot(int value) {  
    if(value < 0)  
        throw new IllegalArgumentException("Negative value!");  
    int root = 0;  
    while(root*root <= value) {  
        ++root;  
    }  
    return root - 1;  
}
```

- If **value** is negative, an **IllegalArgumentException** will be thrown

Multiple catches with inheritance

- Because a parent **catch** will catch a child, you have to organize multiple **catch** blocks from most specific to most general:

```
try {
    dangerousMethod();
}
catch (FusionNuclearExplosionException e) {
    System.out.println("Fusion!");
}
catch (NuclearExplosionException e) {
    System.out.println("Nuclear!");
}
catch (ExplosionException e) {
    System.out.println("Explosion!");
}
catch (Exception e) { // Don't do this!
    System.out.println("Some arbitrary exception!");
}
```

Practice

Sample Question 1

- Write a **LonelyGoat** class that implements the following interface with a **yodel ()** method that returns the lyrics concatenated with itself repetitions times
- For example, values of "**Yodelay!**" and 3 would return:
 - "**Yodelay!Yodelay!Yodelay!**"

```
public interface Yodelable {  
    String yodel(String lyrics, int repetitions);  
}
```

Sample Question 2

- Write a (non-abstract) **Salmon** class that extends the **Fish** class
- Its species name should be "**salmon**"
- Implement the **swim()** method however you want

```
public abstract class Fish {
    private String species;
    public Fish(String species) {
        this.species = species;
    }
    public String getSpecies() {
        return species;
    }
    public abstract String swim();
}
```

Sample Question 3

- What would the following code print out?

```
try {
    System.out.println("Let's eat some week-old scallops!");
    throw new FoodPoisoningException();
    System.out.println("That went great!");
}
catch (NullPointerException e) {
    System.out.println("Null!");
}
catch (FoodPoisoningException e) {
    System.out.println("Barf!");
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index!");
}
finally {
    System.out.println("Tomorrow is another day.");
}
```

Extended programming practice

- We can imagine a hierarchy of inheritance starting with a **Person** with the following members:
 - Name (final)
 - Age
- **Student** extends **Person** and adds:
 - Major
 - GPA
- **Politician** extends **Person** and adds:
 - Political party
- **OtterbeinStudent** extends **Student** and adds:
 - ID number (final)
- Members should have getters and setters as appropriate
- All classes should override the **toString ()** and **equals ()** methods

Upcoming

Next time...

- Exam 1!
 - Monday, in class

Reminders

- **Finish Project 1**
 - **Due tonight!**
- Review chapters 1 – 6, 8 – 12, and 17
- Look over labs, quizzes, and the project to prepare