Week 3 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Inheritance examples
- Overriding methods

Questions?

Project 1

Inheritance Examples

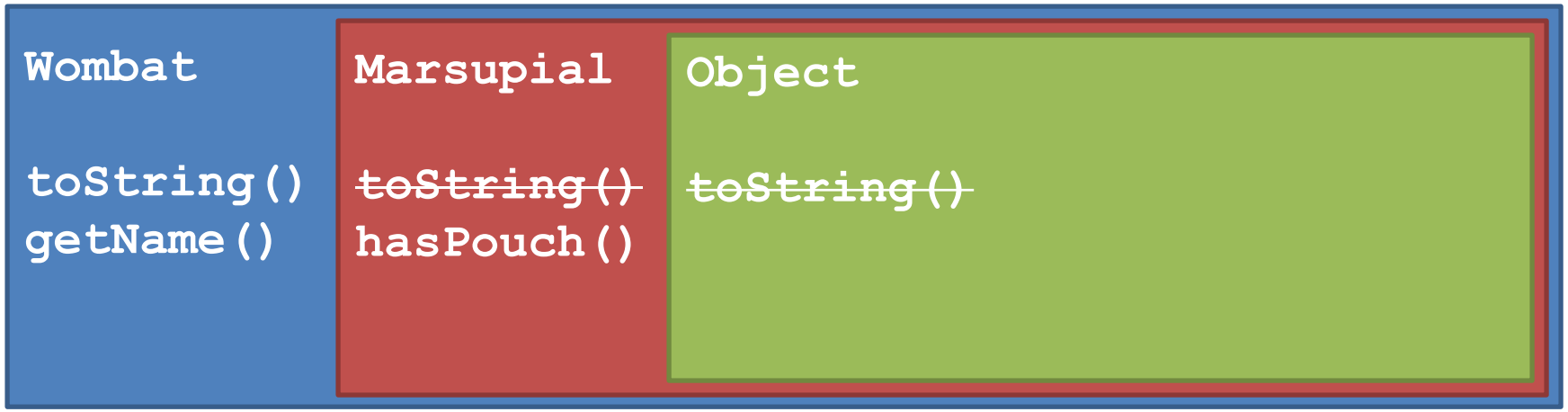
The `Person` class

- 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

Dynamic Binding

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



Weaknesses of dynamic binding

- Dynamic binding is safe and flexible
  - You always get the most up-to-date method
- But it has a performance penalty
- The method that will be called is not known at compile time
- Instead, it has to be looked up in a table inside the object
  - Virtual dispatch table
- To avoid this penalty, languages like C# and C++ require the parent method to be marked as **`virtual`** if you want to override it and get dynamic binding

Static methods

- Static methods **do not** exhibit dynamic binding
- They can be hidden by child classes, but calling them with a class name (or even by a reference) will call the method belonging to the **type** (not the object)

Two classes

```
public class A {
  public static String getMessage() {
     return "A";
  }
}
```

```
public class B extends A {
  public static String getMessage() {
     return "B";
  }
}
```

Static method example

- When a static method is called, it's always based on the reference type, **not** the object type

```
A a = new A();
A b = new B();
System.out.println(A.getMessage()); // "A"
System.out.println(B.getMessage()); // "B"
System.out.println(a.getMessage()); // "A"
System.out.println(b.getMessage()); // "A"
```

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

Why would we want to make a class `final`?

- From a design perspective, all classes should fall into two categories:
  - Classes that you **expect** to be extended
  - Classes that should **not** be extended
- Classes that you expect to be extended might have abstract methods and protected members, and they should be well-documented for someone who wants to extend them
- Most classes will never be extended, so making them **`final`** sends a clear message

`final` method example

- In this class, it's impossible to override the **`reverse()`** method

```
public class Message {
  private String text;
  public Message(String text) {
     this.text = text;
  }
  public String getText() {
     return text;
  }
  public final void reverse() {
     String reversed = "";
     for(int i = 0; i < text.length(); ++i)
           reversed += text.charAt(text.length() - i - 1);
     text = reversed;
  }
}
```

final class example

- This class can never be extended

```
public final class Popeye {
  private boolean spinach = false;
  public boolean hasSpinach() {
     return spinach;
  }
  public void giveSpinach() {
     spinach = true;
  }
  public String toString() {
     return "I am what I am.";
  }
}
```

Abstract Classes

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();
}
```

Extending abstract classes

- A class that extends an abstract class must implement all of its abstract methods (or it will also have to be abstract)

```
public class Rectangle extends Polygon {
   private double length;
   private double width;
   public Rectangle(double length, double width) {
       super(4);
       this.length = length;
       this.width = width;
   }
   public double getArea() {
       return length * width;
   }
   public double getPerimeter() {
       return 2.0*length + 2.0*width;
   }
}
```

Using abstract classes

- You **can** declare a variable with the type of an abstract class
- However, you **can't** instantiate an object of that type
- You can use the variable to store instances of subclasses

```
Polygon polygon = new Rectangle(3.4, 7.1);
System.out.println(polygon.getSides()); // 4
Polygon triangle = new Polygon(3); // Compiler error
```

- Abstract classes provide more functionality than interfaces
  - But you can only extend one abstract class while you can implement an unlimited number of interfaces

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!");
```

UML

UML

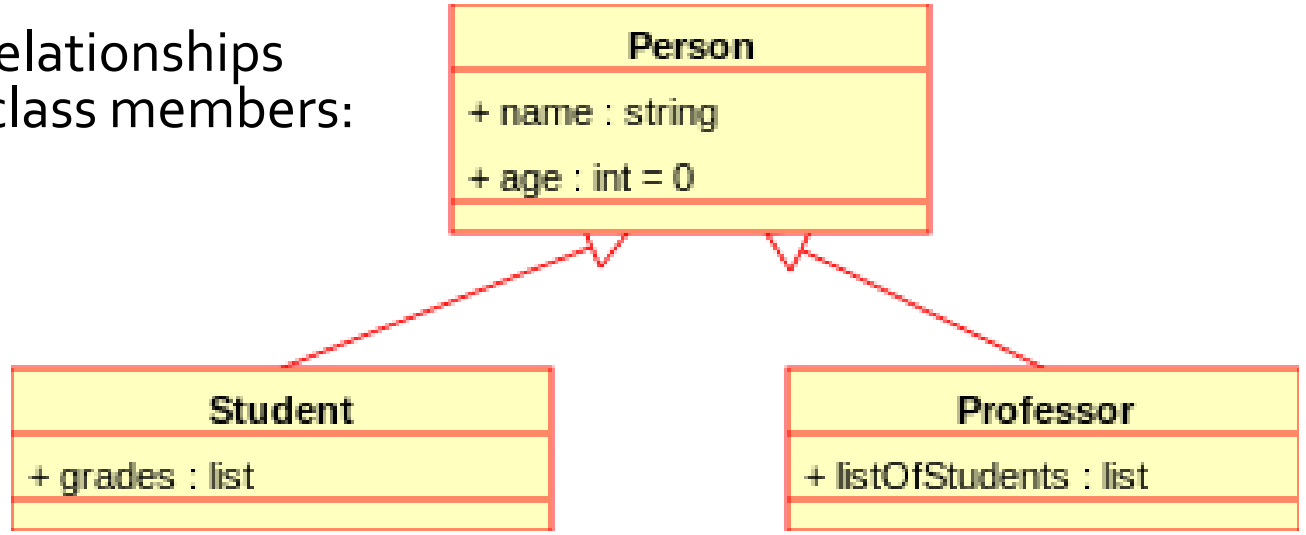
- The **Unified Modeling Language** (UML) is an international standard for graphical models of software systems
- UML was developed in the mid 90s and adopted as an ISO standard in 2005
- UML diagrams can be divided into structural, behavioral, and interaction categories (though interaction is really a subset of behavioral)
- A few useful kinds of diagrams:
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - Class diagrams
  - State diagrams

Class diagrams

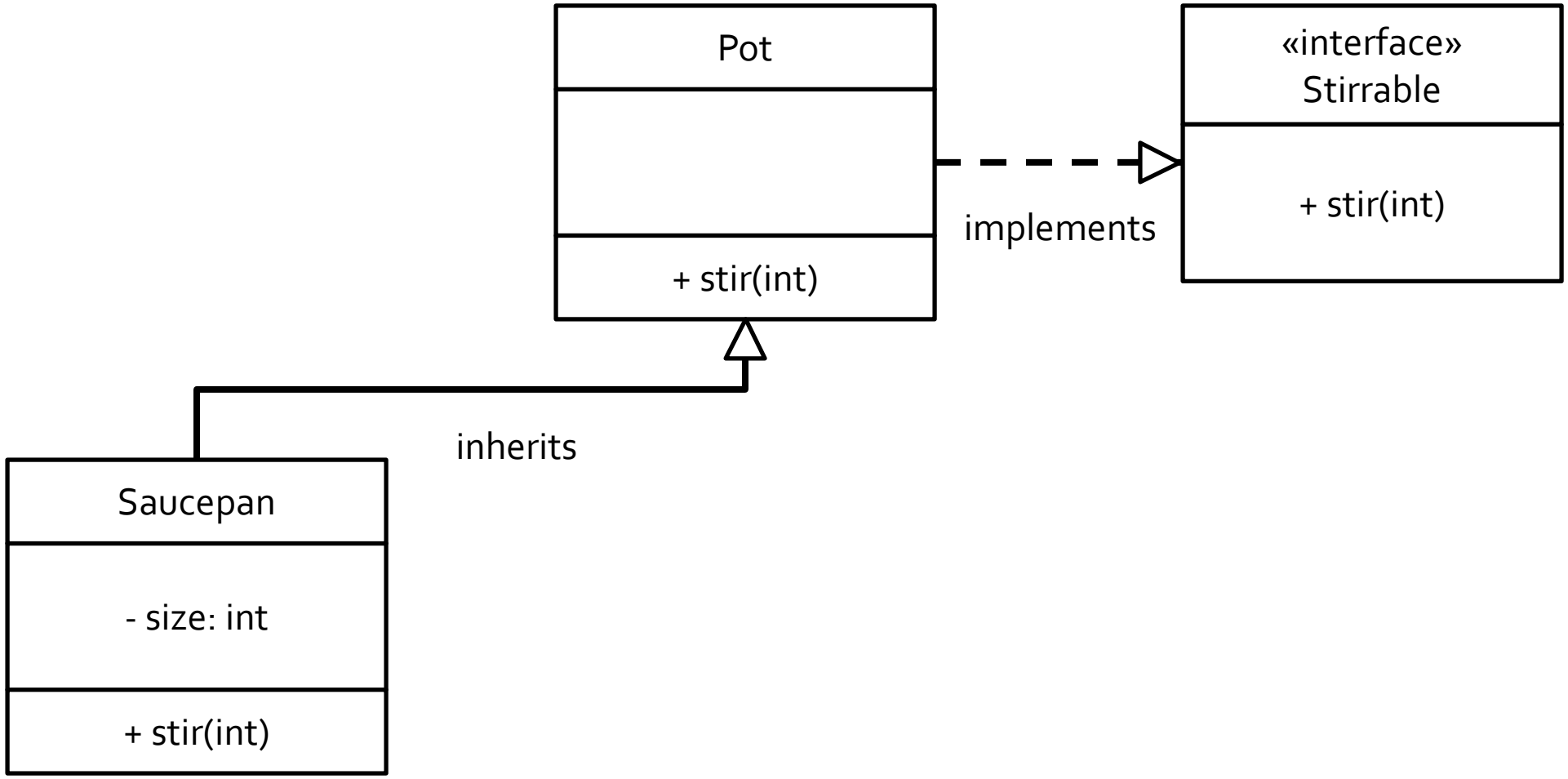
- Class diagrams are commonly used to describe inheritance hierarchies in Java
- Despite their name, they are also used to model tables in databases and other entities
- For inheritance,
  - Arrows point from children up to parent classes
  - Arrows with dashed lines point from classes to interfaces they implement
- Classes in class diagrams often have three parts:
  - Top: Name
  - Middle: Members
  - Bottom: Methods

Class diagrams

- Class diagrams show the different object classes and the relationships between them
- These diagrams often show inheritance relationships
- The following symbols are used to mark class members:
  - **+** Public
  - **–** Private
  - **#** Protected
  - **/** Derived
  - **~** Package
  - ***** Random
- Example from Wikipedia:
- Relationships can also be association, implementation, dependency, aggregation, and composition and can be many to one, one to one, many to many, etc.



Class diagram example



Upcoming

Next time…

- Exceptions

Reminders

- **Michael Thornton talk:**
  - **How to get a Software Engineering Job**
  - Tuesday, February 4, 4-6 p.m.
  - The Point 113
- Read Chapter 12
- Keep working on Project 1
  - Due next Friday