GCIS-124
Software Development & Problem Solving

3: Inheritance

RIT | Golisano College of Computing and Information Sciences

SUN	MON (1/29)	TUE	WED (1/31)	THU	FRI (2/2)	SAT
	Unit 2: Java Classes		Unit 3: Inheritance			
	Assignment 2.1 Due (start of class)	You Are Here	Unit 2 Mini Practicum Assignment 2.2 Due (start of class)			
SUN	MON (2/5)	TUE	WED (2/7)	THU	FRI (2/9)	SAT
	Unit 3: Inheritance		Unit 4: Graphical User Interfaces			
	Assignment 3.1 Due (start of class)		Unit 3 Mini-Practicum Assignment 3.2 Due (start of class)		***Midterm Exam 1*** will be on ***Wednesday, February 14th***.	

3.0 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.

- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should create a `SoftDevII` directory if you have not done so already.
  - Navigate to your `SoftDevII` directory and clone the new repository there.
- Open your repository in VSCode and make sure that you have a terminal open with the **PROBLEMS** tab visible.

Asking for Help

Could be Improved

I don't understand part 4.c. on the homework.

You have all the resources that you need to solve the problems that have been given to you, but sometimes you may not be able to find the answers to the problems that you encounter.

Asking for help in this course is both ***expected*** and ***encouraged***.

Spending time trying to solve the problem yourself will be a more valuable learning experience in the long run.

If you ***do*** ask for help, try to be as detailed as possible to make it easier for others to help you by providing as much detail as you can.

MUCH Better

Part 4.c. on homework 2.2 asks us to write a function that prompts the user to enter two floating point values and to print the product.

I reviewed slides 17-18 in the lecture, and I finished the practice activity. That all works fine.

But when I try to use the values entered by the user, I am getting an "input mismatch" error. Does anyone have a suggestion?

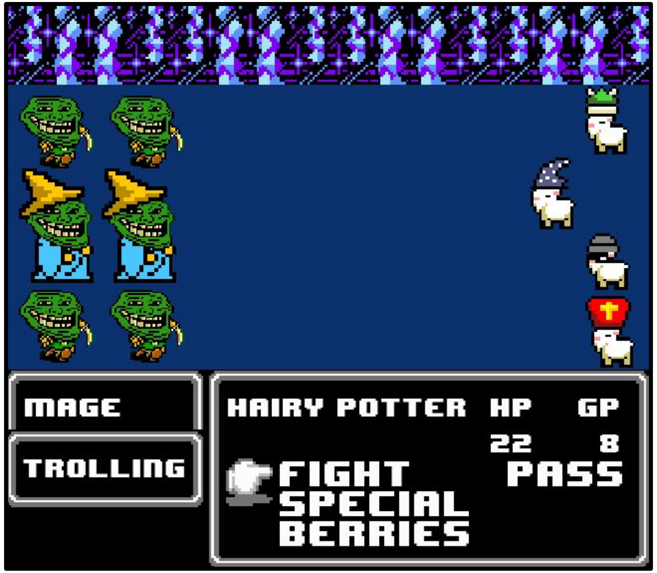
1. Begin by reviewing the lecture slides related to the problem that you are trying to solve
2. Try to solve the related class activities without looking at the answer
3. When asking for help, try to be as specific as possible about the problem you are having

Inheritance

We'll spend most of this unit using different types of ***inheritance*** to implement an RPG based on the classic *Goats vs. Trolls* CCG.

- In the previous unit, we learned how to create classes in Java that include:
  - State (Fields)
  - Behavior (Methods)
  - Data privacy (Access Modifiers)
  - Constructors
  - Static and non-static state and behavior
- During this unit we will learn about a very powerful feature of ***object-oriented programming***: ***inheritance***.
  - Subclassing
  - Overriding Methods
  - Polymorphism
  - Abstract Classes
  - Interfaces
- Today we will specifically focus on ***reuse*** through subclassing.
  - One class ***inherits*** the state and behavior defined in another class.
  - The new class can then modify existing behavior or add new state and/or behavior.

Goats vs. Trolls V: The Trolls Strike Back



As the screen above shows, GvTV will leverage modern hardware to enable its cutting edge graphics.

The *Goats vs. Trolls* CCG has been so wildly successful that the powers that be have decided to expand the IP into a new video game inspired by the classic JRPGs of the 1980s. Your software development company has been contracted to develop the highly anticipated game. Goats vs. Trolls: The RPG. The player will control a party of 4 goats, as they battle against endless waves of enemy trolls. Your team will begin its design by focusing on the heroic Goats.

Type	Hit Points	Attack	Special
Mage	120	Magic Missile 9 (Magic Damage) hits 4 times	+25% physical damage taken -25% magical damage taken
Fighter	150	Cleave 25 (Physical Damage)	-25% physical damage taken +25% magical damage taken

3.1 Simple Domain Analysis

Domain analysis is the activity of identifying and documenting the commonalities and variabilities in related software systems. Begin your design with a simple domain analysis.

Things to Note:

- All goats have a **name**, e.g. “Hairy Potter.”
- All goats can be **healed** for some of their hit points.
- If a goats current HP drops to 0, the goat is knocked **unconscious**.

Class			
State			
Behavior			

3.1 Simple Domain Analysis

Domain analysis is the activity of identifying and documenting the commonalities and variabilities in related software systems. Begin your design with a simple domain analysis.

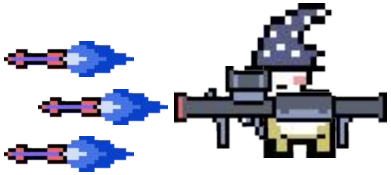
Things to Note:

- All goats have a **name**, e.g. "Hairy Potter."
- All goats can be **healed** for some of their hit points.
- If a goats current HP drops to 0, the goat is knocked **unconscious**.

Class	ATTACK	MAGE	FIGHTER
State	NAME # OF HITS DAMAGE TYPE	NAME MAX HP CURRENT HP	NAME MAX HP CURRENT HP
Behavior	GETTERS	ATTACK ATTACKED HEAL IS CONSCIOUS?	ATTACK ATTACKED HEAL IS CONSCIOUS?

3.2 Damage Types

An attack in GvT may do one of several predefined types of damage. Define a Java enumeration that can be used to represent the different damage types in a game of GvT.



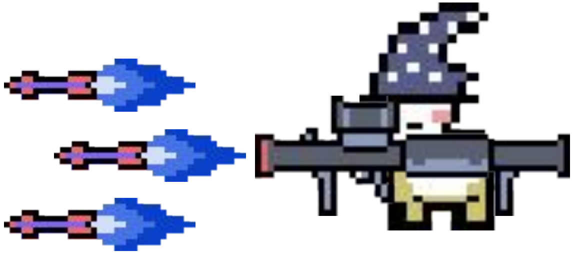
Goat Mage's *Magic Missile* attack does *Magical* damage...

...while Goat Fighter's *Cleave* attack does *physical* damage.

- Create a new package under `unit03` named "`gvt`". You will use this package for all of the code that you write today.
- Create a new Java enum named "`DamageType`". Enumerate the possible values for damage types including:
  - Physical
  - Magical
  - Poison
  - Holy
  - Elemental
- There is no need to test your new `enum`; you will be using it in the next activity.

3.3 Plan of Attack

Attacks in GvT are too complex to be represented by a simple primitive type. Each attack has a name, a number of hits (each of which has its own damage amount), and a damage type. Create a new class to represent an attack.



The Mage's primary attack is *"Magic Missiles."* It hits *4 times* for *9 points* of *magical damage*.

- Create a new Java class named "`Attack`". An attack includes:
  - A ***name***, e.g. "Magic Missiles".
  - A variable number of ***hits***, each of which is an integer value (hint: use an array).
  - A ***damage type***.
- Create a ***constructor*** that sets all three fields.
- Create an ***accessor*** for each of the fields.
- Once assigned, none of the values in an attack should ever change, so there is no need for any ***mutators***.

3.4 A Goat Mage

A professional software developer will design the software that they write ***first*** and then implement their design. Software developers must be able to read UML class diagrams and implement classes based on the design. Use the UML class diagram to begin implementing the Mage class.

Mage

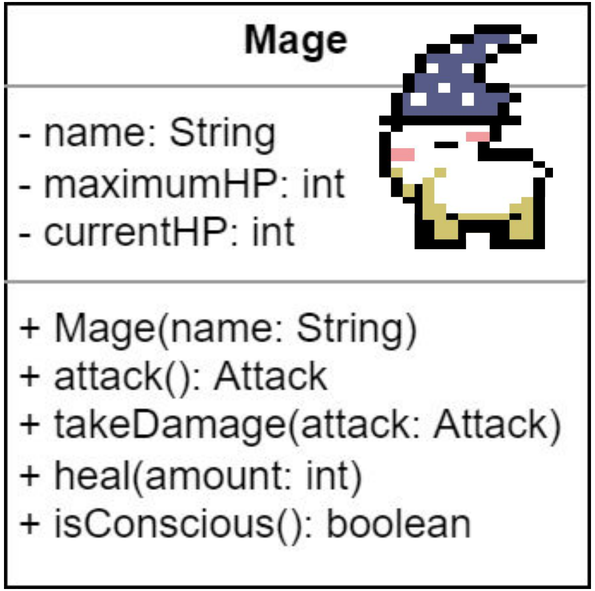
- name: String
- maximumHP: int
- currentHP: int

+ Mage(name: String)
+ attack(): Attack
+ takeDamage(attack: Attack)
+ heal(amount: int)
+ isConscious(): boolean

- Create a new Java class named "`Mage`" and begin implementing the class.
  - Add the necessary fields.
  - Create a constructor that initializes the name and sets the current and max HP to the default values (120).
  - Add a `toString()` that returns a String in the format `"A mage named Hairy Potter with 120/120 hit points!"`
- Add a `main` method with the appropriate signature.
  - Create two instances of your new class.
  - Print the mages to standard output.

3.5 The Goat Mage (continued)

The Goat Mage is still missing some essential functionality. Mages can create Magic Missile attacks, can be attacked by other characters, and can be knocked unconscious. Finish implementing the Mage class now.



Mage

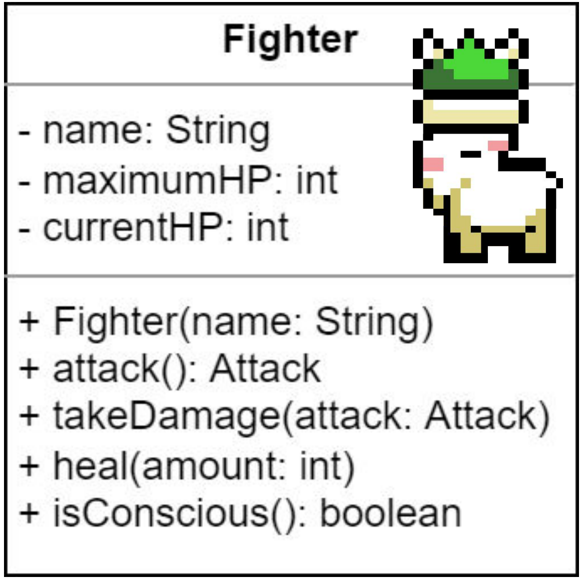
- name: String
- maximumHP: int
- currentHP: int

+ Mage(name: String)
+ attack(): Attack
+ takeDamage(attack: Attack)
+ heal(amount: int)
+ isConscious(): boolean

- Use the UML as your guide to implement the remaining methods in the Mage class. Remember the following:
  - They can ***attack*** with ***"Magic Missiles,"*** hitting ***4 times*** for ***9 points*** of ***magical*** damage.
  - They can ***heal*** for a specified number of hit points, but not above the max HP.
  - They may be ***unconscious*** if the current HP is at 0.
  - They can ***take damage*** from an attack which reduces their current HP, but not below 0. Remember that ***magical damage*** is reduced by 25% but ***physical damage*** is increased by 25%.
- Use the `main` method to test your class.
  - Have your mages attack each other.
  - Print the mages to standard output after the attacks.

3.6 The Goat Fighter

GvT wouldn't be a very fun game with only one character class! Let's implement a new class to represent Goat Fighters.

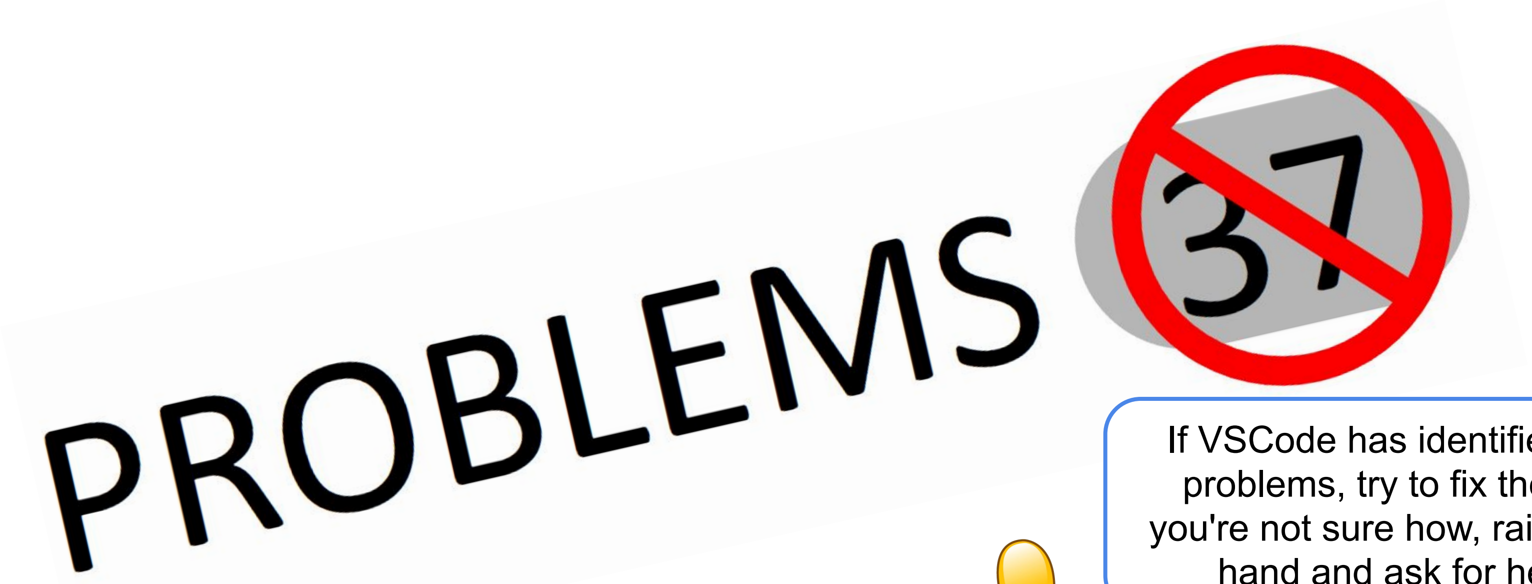


Fighter

- name: String
- maximumHP: int
- currentHP: int

+ Fighter(name: String)
+ attack(): Attack
+ takeDamage(attack: Attack)
+ heal(amount: int)
+ isConscious(): boolean

- Create a new Java class named "`Fighter`" and begin implementing the class. ***Hint***: begin by copy/pasting your `Mage` code.
- Use the UML as a guide.
  - Add fields and a constructor. The maximum HP should be 150.
  - Add the `toString()` method.
- When implementing the Fighter's ***behavior***:
  - They can ***attack*** with ***"Cleave,"*** hitting ***once*** for ***25 points*** of ***physical damage***.
  - They can ***heal*** for a specified number of hit points, but not above the max HP.
  - They may be ***unconscious*** if the current HP is at 0.
  - They can ***take damage*** when from an attack which reduces their current HP, but not below 0. Remember that ***magical damage*** is increased by 25% but ***physical damage*** is reduced by 25%.
- Add a `main` method with the appropriate signature.
  - Create two instances of your new class.
  - Print both to standard output.
  - Have your fighters attack each other, and print them to standard output after the attacks.



Whenever you see this image, it's a reminder to check your PROBLEMS tab.

If VSCode has identified any problems, try to fix them. If you're not sure how, raise your hand and ask for help!

And remember: it's not about *avoiding* making mistakes. It's about learning how to fix them as we make them.

3.7 Goat Fight Club

PROBLEMS ~~37~~

Now that we have a couple of goat character classes, let's put them to the test by creating an arena in which they can do battle!

Two goats enter. One goat leaves.

- Create a new Java class named "`GoatArena`" and define a `static` method named "`battle`" that declares parameter for `goat1` (a `Mage`) and `goat2` (a `Fighter`).
- Have the two goats engage in an epic battle!
  - As long as both goats are conscious, have each goat attack the other in turn.
  - To be fair, both goats will always get to attack the other.
  - Be sure to print attacks and the status of each goat after each round.
  - Once the battle is over, determine the winner by checking to see which goat is still conscious.
- Define a `main` method with the appropriate signature.
  - Make one of each type of goat and send them to battle.

3.8 Mage Fight?

PROBLEMS 37

Who says that it always has to be a mage vs. a fighter in the arena? What if we wanted two mages to fight each other? Or what about two fighters?

At this point, we have ***a lot*** of duplicated (or *nearly* duplicated code).

What happens if we add a Thief class? Or a Cleric? Or a Paladin?

- Open the "`GoatArena`" and overload the "`battle`" method so that it declares parameters for `goat1` and `goat2` (both mages).
    - To speed things up, copy and paste the code from the other battle method.
    - If you make sure that the parameter names are the same, how much of the code do you need to change?
- Use `main` to call the new battle function, this time with two mages.
- Repeat the entire process, but this time both `goat1` and `goat2` should be fighters.

Reuse

DRY

Don't Repeat Yourself

DRY is a driving principle of software engineering that will be continuously emphasized.

Up until now, we have used ***functions*** to reuse code, but that technique will ***not work*** if two or more classes need the same functionality.

When using classes and objects, identical functionality in two or more classes may indicate an opportunity to leverage ***inheritance***.

- One of the major benefits of ***object-oriented programming*** is ***reuse***.
  - Write code ***once***, and use it in lots of places.
- Reuse is an alternative to ***"copy and paste" coding***, which ***duplicates*** the same code wherever it is needed.
  - There are many drawbacks to duplicating code, not the least of which is that it is ***wasteful***, ***inefficient***, and duplicates ***bugs***.
- Up to this point, the primary mechanism for ***reuse*** has been ***functions***, but what if two ***different*** classes need the ***same*** functionality?
  - What if two or more classes implement the ***same behavior*** in exactly the same way?
  - What if two classes have ***identical state***?
- Object-oriented programming provides a core feature that allows different classes to ***reuse*** code: ***inheritance***.
  - One class may ***extend*** another class to inherit its accessible ***state*** and ***behavior***.
  - This creates a ***parent/child*** relationship between the two classes.

3.9 Identifying Common Attributes

We will be creating a new ***parent*** class to contain all of the fields and methods that our two goats have in common. Let's start by identifying what those are!

You may find it useful to open the two classes side-by-side in your editor (i.e. split to the right).

	Common to Both	Unique to Mage	Unique to Fighter
State			
Behavior			

3.9 Identifying Common Attributes

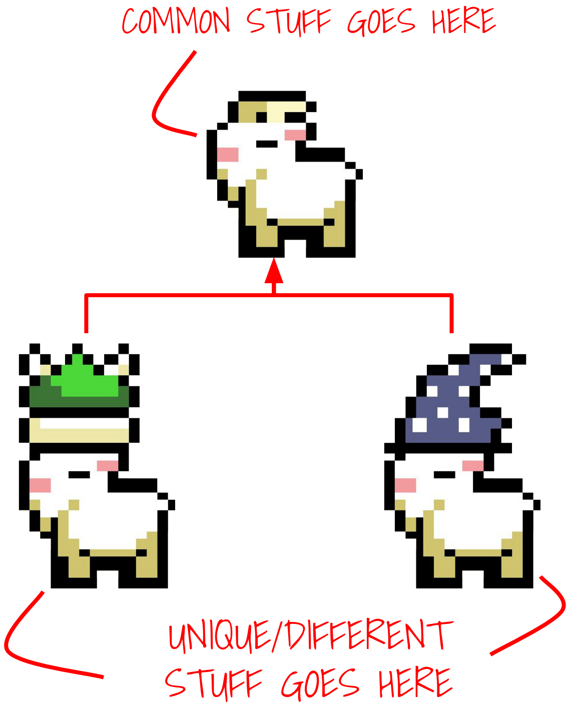
We will be creating a new ***parent*** class to contain all of the fields and methods that our two goats have in common. Let's start by identifying what those are!

You may find it useful to open the two classes side-by-side in your editor (i.e. split to the right).

	Common to Both	Unique to Mage	Unique to Fighter
State	NAME MAX HP CURRENT HP	MAX_HP	MAX_HP
Behavior	CONSTRUCTOR ACCESSORS HEAL IS CONSCIOUS	ATTACK TAKE DAMAGE	ATTACK TAKE DAMAGE

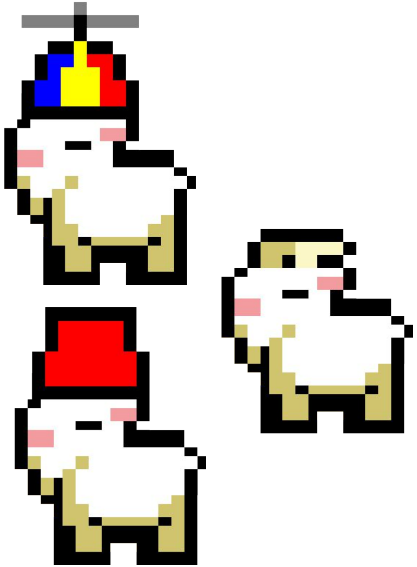
A Common Goat

- The Mage and Fighter classes share all of the same ***state***:
  - Name, Max HP, Current HP
- Both classes also implement some of the exact same ***behavior***:
  - Very similar constructors.
  - Accessors for name, current HP, & Max HP.
  - Methods for healing and consciousness.
- We will create a class that encapsulates all of the common state and behavior so that the Mage and Fighter classes can ***reuse*** it.
  - We'll see exactly how to do this presently.
- The process of identifying common state and behavior and putting it into a parent class is called ***creating an abstraction***.



3.10 A Common Goat

Inheritance allows one class to reuse the accessible state and behavior defined by another class so that we don't need to copy-and-paste it in multiple places. Let's create a new Goat class that contains all of the common attributes of the Mage and Fighter classes.

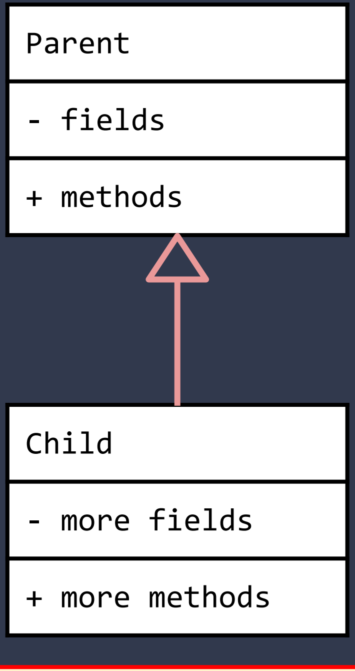


- Create a new Java class named "`Goat`" and use it to encapsulate all of the common state and behavior between your two existing goat classes.
  - ***Copy/paste*** all of the code from one of the two goat classes.
  - Update the name of the constructor.
- How should you handle the ***attack*** and ***take damage*** methods?
  - All goats ***need*** both of these methods.
  - But there is no common implementation.

Not all heroes wear capes. And not all goats wear hats.

Inheritance

- A child class ***inherits*** the ***accessible*** state and behavior of its ***parent class***.
  - ***Private*** members are not inherited.
  - Child classes ***do*** inherit ***protected*** members.
  - ***Constructors*** are ***not*** inherited. The child class must define its own.
- This means that an instance of the child class can reuse the state and behavior in its parent.
  - This includes ***accessible*** fields in the parent class.
  - This also includes any ***accessible*** methods in the parent class.
- Inherited state and behavior can be accessed ***through the child*** using dot notation.
  - This also means that an instance of the child class can be used anywhere the parent is expected.
  - For example, an instance of the ***child class*** may be passed as an argument to a method that declares a parameter of its ***parent's type***. Code inside the method can use ***any*** of the state and behavior defined by the ***parent***.
  - This is a core object-oriented concept known as ***polymorphism***.
- In Java, a child class uses `extends` to establish an inheritance relationship with a parent class.
  - e.g. `public class Child extends Parent`



Remember, arrows in UML indicate the direction of dependency. Because the child depends on the parent (and not the other way around), inheritance in UML is depicted with a ***closed arrow*** (that is not filled) that points ***from the child to the parent***.

`super` & Access Modifiers

- We already know that an object uses `this` to refer to itself and use its ***own*** state and behavior. It can be used in several different ways, e.g.:
  - Setting fields: `this.name = name;`
  - Chaining constructors: `this(name, age);`
  - Invoking methods: `this.getName();`
- Similarly, an object uses `super` to access state and behavior defined by its parent class, e.g.
  - Invoke a constructor: `super(x, "white");`
  - Access fields: `return super.name;`
  - Calling methods: `super.getAge();`
- Constructors are ***not*** inherited from the parent class; the child must define its own ***and*** call one of the parent constructors using `super`.
  - The exception is that if the parent class has a ***default*** or ***parameterless constructor***, in which case it will be invoked ***transparently*** by default if no other constructor is explicitly called.

A child class can only use `super` to access ***accessible*** state and behavior in its parent class. Accessibility to the child is determined by the access modifier used with each field, method, and constructor.

What has access?	`private`	none (*package private*)	`protected`	`public`
Instances of the same class	✔	✔	✔	✔
Classes in the same package		✔	✔	✔
Child Classes			✔	✔
Everything				✔

Inheritance Example

```java
public class Animal {
    private String name;
    private double weight;

    public Animal(String name, double weight) {
       this.name = name;
       this.weight = weight;
    }

    public String toString() {
       return "Animal[name=" + name
          + ", weight=" + weight + "]";
    }

    public String getName() {
       return name;
    }

    public void greet(Animal o) {
       System.out.println(name + " greets "
          + o.name + "!");
    }
}
```

Child classes use `extends` to create an ***is-a relationship*** with a parent class.

Constructors are ***not*** inherited. A child class must define its own constructors and call one of the parent constructors using `super`.

Child classes ***cannot*** directly access `private` state or behavior in the parent class. Accessors and mutators will need to be used.

Through subclassing, `Rabbit` "is a" `Animal`, and can be used with any code written to use `Animal`. ***Polymorphism***!

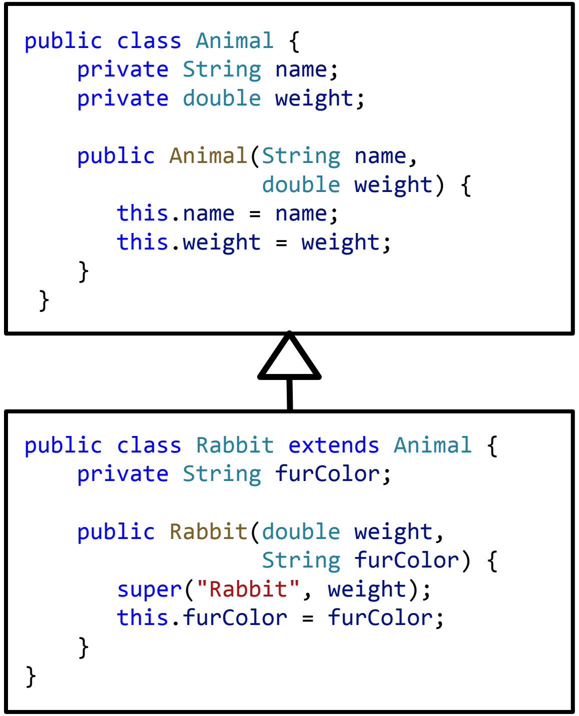
```java
public class Rabbit extends Animal {
    private String furColor;

    public Rabbit(double weight,
                  String furColor) {
       super("Rabbit", weight);
       this.furColor = furColor;
    }

    public void move() {
       System.out.println(~~super.name~~
          + " goes hop, hop, hop!");
    }
 }
```

Use getName() instead

```java
Rabbit r = new Rabbit(2.5, "brown");
Animal a = new Rabbit(3, "white");

a.greet(r);
```

3.11 Begin Refactoring Mage

Now that you have encapsulated all of the common state and behavior in the `Goat` class, refactor `Mage` so that it extends `Goat`.



```java
public class Animal {
    private String name;
    private double weight;

    public Animal(String name,
                  double weight) {
       this.name = name;
       this.weight = weight;
    }
 }
```

```java
public class Rabbit extends Animal {
    private String furColor;

    public Rabbit(double weight,
                  String furColor) {
       super("Rabbit", weight);
       this.furColor = furColor;
    }
}
```

- Open your `Mage` class and modify it so that it subclasses your common `Goat`. For now we will focus only on the ***constructor***.
  - Begin by ***commenting all but the*** `main` ***method*** in the body of the class.
  - Use `extends` to establish the inheritance relationship with `Goat`.
- Try to ***compile*** and ***run*** the class.
  - Compilation will ***fail*** because you have not yet defined a ***constructor***.
  - Add a constructor to your `Mage` class that uses `super` to call the constructor in the parent class.
  - What ***parameters*** will your `Mage` constructor need? ***Hint***: some values never change from one `Mage` to the next.
- ***Compile*** and ***run*** the class again.
  - Does the `main` method still work the way that you expect it to?

Overriding Methods

The `toString()` method in the `Animal` class returns a `String` that doesn't include all of the detail about a `Rabbit` (i.e. its fur color is missing).

The Rabbit class may ***override*** the implementation of the method in the Animal class by declaring a ***new implementation*** with the ***exact same signature***.

```java
public class Rabbit extends Animal {
   // other fields, constructors, and methods
   // are not shown here for brevity

   @Override
   public String toString() {
      return "Rabbit[weight=" + getWeight()
         + ", fur color=" + furColor + "]";
   }
}
```

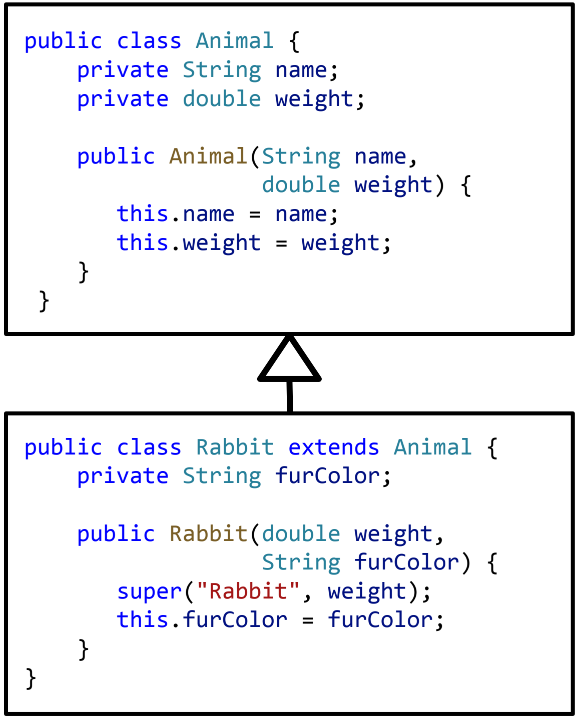
If the ***optional*** `@Override` annotation is used, the Java compiler will validate the method signature.

If the method is called on an instance of the `Rabbit` class, ***its version*** will be used ***instead*** of `Animal`'s.

- Inheritance enables ***polymorphism***, which means that any code written to use the ***parent*** will also work with ***any*** of its ***children***.
    - This is because the child class ***inherits*** all of the accessible state and behavior from its parent.
    - This means that ***any*** code that uses state and behavior defined in the parent class will find the same state and behavior in the child as well.
- Conversely, this means that state and behavior that is ***not*** defined in the parent, ***cannot*** be used via polymorphism.
    - If the method ***only*** exists in the ***child class***, then it ***can't*** be used through a variable of the ***parent type***.
    - This means that we sometimes define methods in the parent to guarantee that ***all*** children will include them.
- In these cases, the child may ***modify*** or ***replace*** the behavior that is defined in its parent.
    - The child class ***overrides*** the method in the parent class by defining a new method with the ***exact same signature***.
    - If the method is called on an instance of the child class, the ***child's version*** is used instead of the parent's.
    - If necessary, the child can call the parent's version suing `super`, e.g. `super.aMethod();`

3.12 Finish Refactoring Mage

One major benefit of inheritance is that we can eliminate duplicate code. Continue refactoring the `Mage` class to override methods and delete any duplicate code should be inherited from `Goat`.



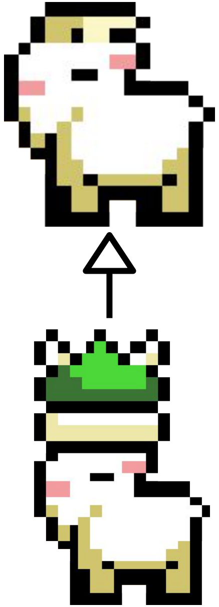
```
public class Animal {
    private String name;
    private double weight;

    public Animal(String name,
                  double weight) {
       this.name = name;
       this.weight = weight;
    }
 }
```

```
public class Rabbit extends Animal {
    private String furColor;

    public Rabbit(double weight,
                  String furColor) {
       super("Rabbit", weight);
       this.furColor = furColor;
    }
}
```

- Open your `Mage` class and examine the code that you have commented out and selectively ***delete*** or ***uncomment*** it.
  - Which fields can safely be ***deleted***?
  - Which methods can safely be ***deleted***?
  - Which methods do you need to keep to ***override*** the same methods in the `Goat` class? Don't forget to use the `@Override` annotation!
- ***Compile*** and ***run*** the class.
  - Does the `main` method work the way that you expect it to?

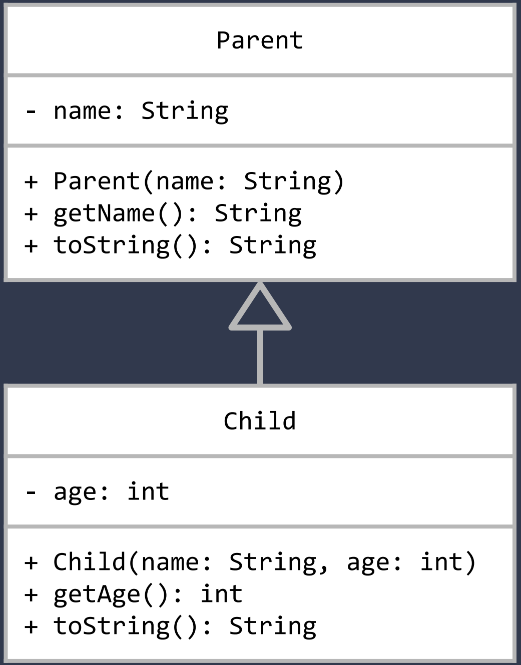
3.13 Refactor Fighter

The `Mage` class is now making effective use of inheritance to reuse the state and behavior from its new parent class. Unfortunately, `Fighter` still has a lot of duplicate code! Let's refactor the `Fighter` class to extend `Goat` now.



- Open the `Fighter` class and modify it so that it subclasses your common `Goat`.
  - Use `extends` to establish the inheritance relationship.
  - You will need to write at least one constructor. Use super to call the parent constructor.
  - Which methods can be safely deleted?
  - Which methods will ***override*** the implementations in the parent class?
- Experiment by running the class. Does the `main` method still work?

Polymorphism



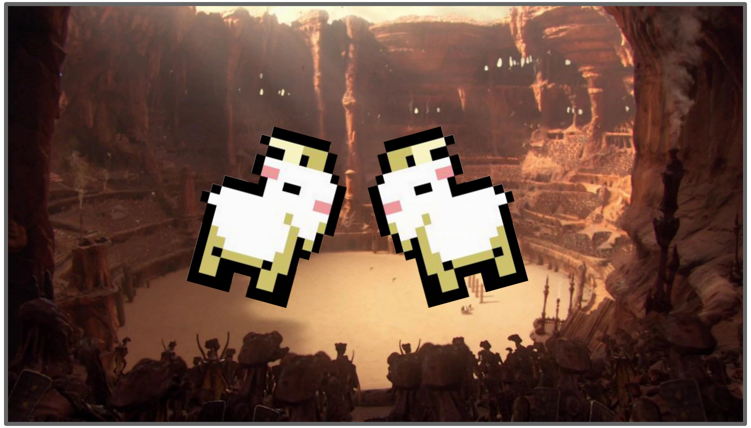
```
Parent
- name: String
+ Parent(name: String)
+ getName(): String
+ toString(): String

Child
- age: int
+ Child(name: String, age: int)
+ getAge(): int
+ toString(): String
```

Polymorphism is a core principle of ***object-oriented programming***. It makes code more powerful, flexible, and reusable. It should be used whenever possible!

- When one class extends another, it ***inherits*** all of the ***accessible*** state and behavior from the parent class.
  - The ***subclass*** can be used with any code that has been written to use the ***superclass***.
  - This is because any code that uses state and behavior defined by the parent will find that same state and behavior in the child.
  - This is a key feature of ***object-oriented programming*** that is known as ***polymorphism***: code written to use a parent will work with any of its children.
- However, there is another equally important aspect of polymorphism: if a child ***is*** used in place of its parent ***and*** the child overrides the methods in the parent, the ***child's version*** of those methods will be used.
  - The ***declared type*** of a variable determines the state and behavior that can be used through that variable.
  - The ***assigned type*** determines the ***implementation*** that is used.
  - For example, `Parent p = new Parent();` will assign an instance of the `Parent` class to the variable, and its methods will be used ***regardless***.
  - But `Parent p = new Child();` will assign an instance of the `Child` class, and ***its*** versions of any ***overridden*** methods will be used instead.

3.14 Goat Fight?

Polymorphism means that we can write code that uses the `Goat` class and it will work with any class that extends `Goat`! This means that we can write ***one*** battle method that declares `Goat` parameters and it will work with ***any*** kind of `Goat`! No more duplication!



Wow! One method that works with any kind of goat?

- Open the "`GoatArena`" and delete all but one "`battle`" method.
    - Change the parameter types to the `Goat` parent class.
- What happened?!

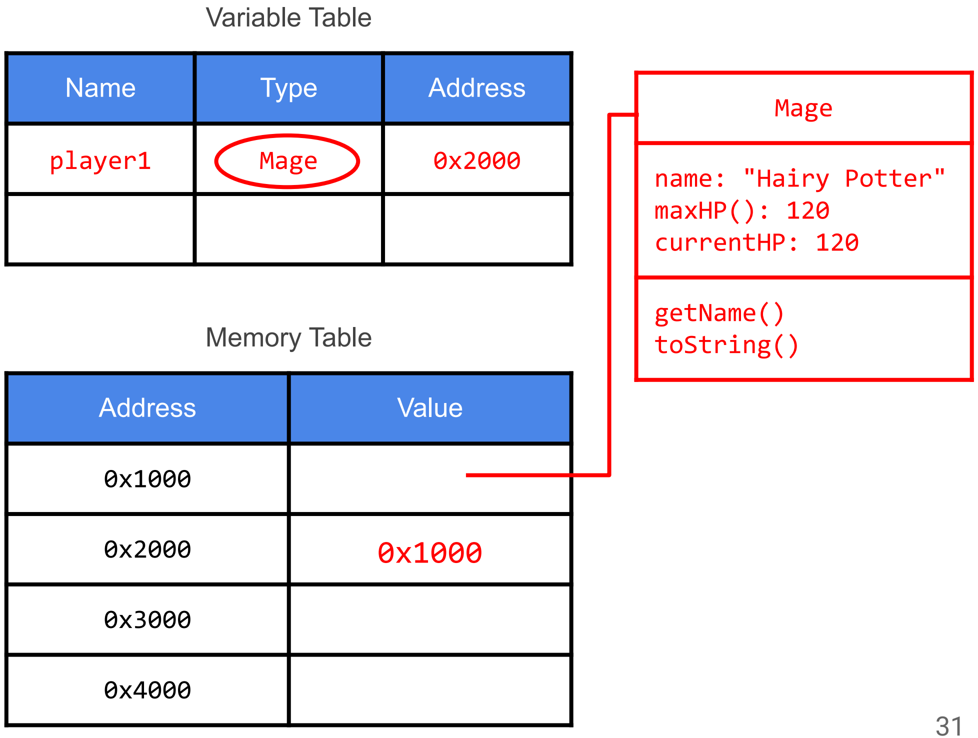
A Closer Look

One way to visualize what is going on in a program is through the use of a ***memory map***.

When a new ***reference type*** is created, the Java Virtual Machine will allocate sufficient ***memory*** to store the object and all of its fields.

Each variable that refers to the object is stored as an entry in the ***variable table*** including its ***name***, ***type***, and it's ***address*** in memory. Note that the value being stored is the address of the object to which the variable refers.

It is the ***declared type*** in the variable table that determines the state and behavior that is accessible through the reference, e.g. the `Mage` class.



Variable Table

Name	Type	Address
`player1`	`Mage`	`0x2000`

Memory Table

Address	Value
`0x1000`	
`0x2000`	`0x1000`
`0x3000`	
`0x4000`	

```
Mage

name: "Hairy Potter"
maxHP(): 120
currentHP: 120

getName()
toString()
```

A Closer Look

When the reference is passed in as the argument for a method parameter, a ***second*** reference to the ***same*** object is created.

The type of the new reference is determined by the declared type of the parameter, e.g. the `Goat` class.

Even though the actual object in memory is a `Mage`, the ***reference type*** (`Goat`) determines the state and behavior that can be accessed through the parameter.

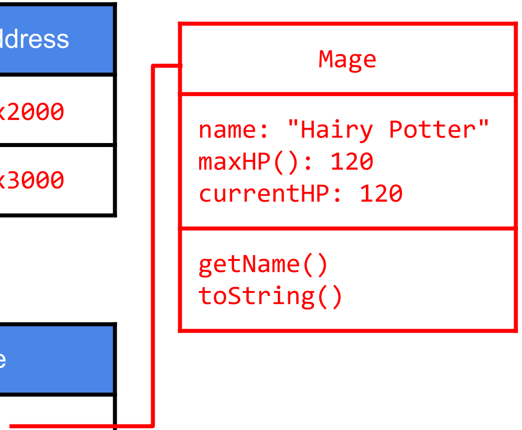
Because the `attack` and `takeDamage` methods are defined in the `Mage` class (and not the `Goat` class), they cannot be used with a `Goat` parameter.

Variable Table

Name	Type	Address
player1	Mage	0x2000
goat1	Goat	0x3000

Memory Table

Address	Value
0x1000	
0x2000	0x1000
0x3000	0x1000
0x4000	



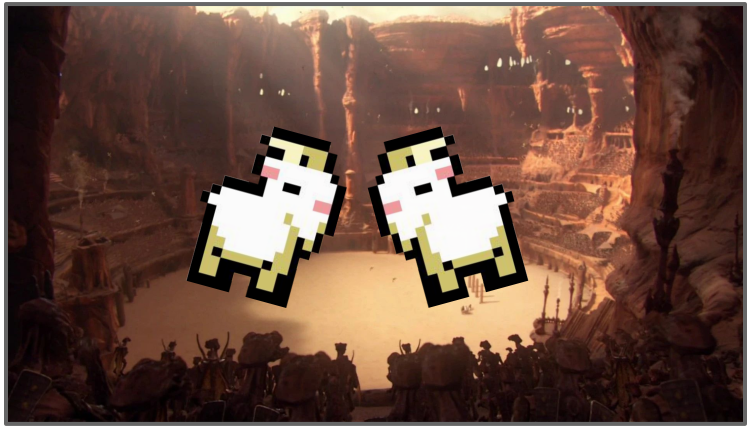
Mage

name: "Hairy Potter"
maxHP(): 120
currentHP: 120

getName()
toString()

3.15 Goat Fight Part 2

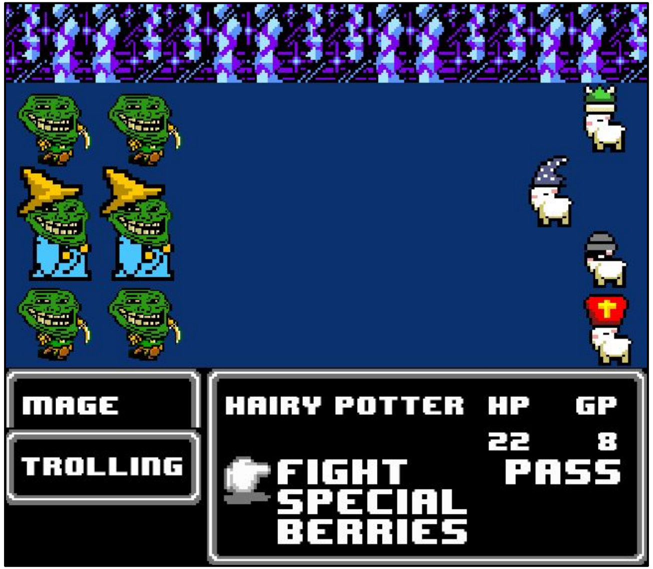
In order for ***polymorphism*** to work the way that we want it to, the ***parent*** class must define ***all*** of the methods that we want to use (even if those methods don't make a heck of a lot of sense). Let's "fix" the `Goat` class so that it contains everything that we need for the specific goat types to battle each other.



Wow! One method that works with any kind of goat?

- Open the "`Goat`" class and stub out the missing methods.
  - The `attack()` method should just return `null`.
  - The `takeDamage()` method should be empty.
- Open your "`GoatArena`" class. Are there any syntax errors?
  - Modify the `main` method to try battling different combinations of goats against each other.
  - What happens if you try to pit a plain `Goat` against another plain `Goat`?

New Goat Classes!



Let's see how flexible polymorphism ***really*** is by creating a couple more goat classes to battle in the arena!

Take a few minutes to think about a Goat class that you would like to play in a game of GvT. If you can't think of one of your own, here are some suggestions.

Type	Hit Points	Attack	Special
 Thief	125	**Stabbity-Stab** 10-20 (Physical Damage) hits 1-3 times; each hit has a 25% chance to crit	-25% Poison Damage Taken
 Cleric	125	**Bell, Book, & Candle** 15-25 (Holy Damage)	Holy Damage *heals* for 25%

3.16 A New Goat

If we make a new class that *also* `extends Goat`, then ***polymorphism*** will allow us to send it to battle in the arena without needing to change any other code. AMAZING.

- Create a new Java class for your new Goat character type.
  - Begin by extending `Goat` right away.
  - Add any fields, constants, and constructors.
  - Implement at least the `attack` and `takeDamage` methods.
- Add a `main` method with the appropriate signature.
  - Create two instances of your new class.
  - Print both to standard output.

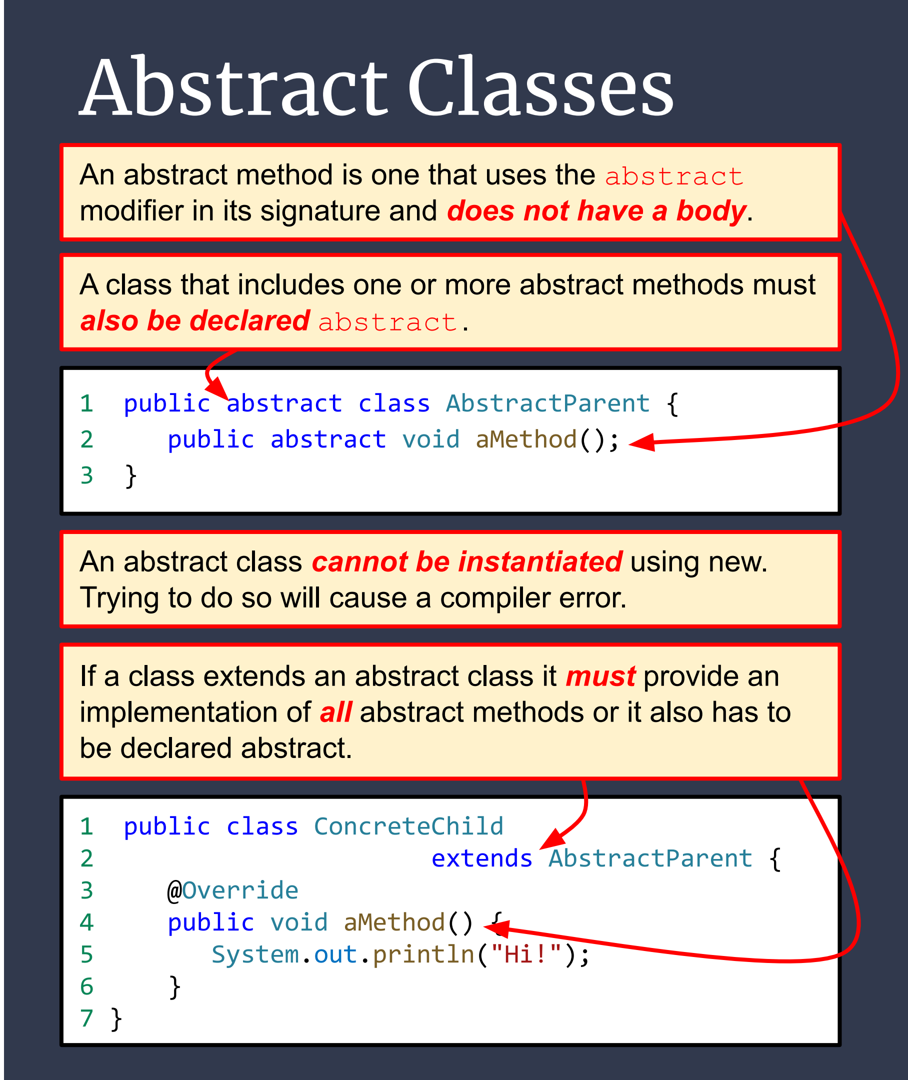
3.17 GOAT RUMBLE!

Your Goat Battle Arena should work with Goat or any of its subclasses. Experiment with different combinations of goat fighters. How many different Goat battles are possible?

Wow! One method that works with any kind of goat?

- Open the "`GoatArena`" and modify the main to wage battles against any combination of Goats that you can think of.
  - Do they all work?

- So hopefully at this point you are beginning to grasp that ***polymorphism*** is important because it allows us to write code ***once*** that will work with lots of different objects as long as they are related by inheritance.
- But this sometimes means that we have to add methods to a parent class that aren't very useful.
  - For example, the `attack()` and `takeDamage()` methods in our `Goat` class.
- In order to guarantee that the methods can be used through a variable of the parent's type, they must be ***defined*** as part of the parent class.
- But the implementations are pointless and never used.
  - ***Every*** Goat subclass overrides ***both*** methods, and so the parent implementations are never used.
- In Java it is possible to ***define*** a method without ***implementing*** it by making it ***abstract***.
  - The method signature includes the `abstract` modifier, e.g. `public abstract Attack attack();`
  - There is ***no implementation*** and ***no method body***.
- A class that includes one or more abstract methods must also be declared abstract.
  - For example `public abstract class Goat {...}`
  - An `abstract` class ***cannot be instantiated*** using new.
  - ***All*** subclasses ***must*** provide an implementation of any `abstract` methods (or be declared `abstract`).



Abstract Classes

An abstract method is one that uses the `abstract` modifier in its signature and ***does not have a body***.

A class that includes one or more abstract methods must ***also be declared*** `abstract`.

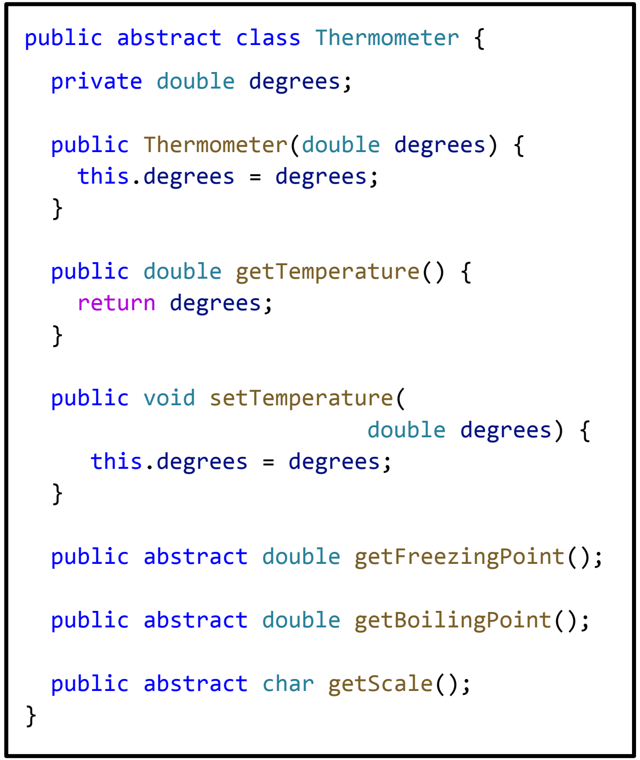
```java
public abstract class AbstractParent {
   public abstract void aMethod();
}
```

An abstract class ***cannot be instantiated*** using new. Trying to do so will cause a compiler error.

If a class extends an abstract class it ***must*** provide an implementation of ***all*** abstract methods or it also has to be declared abstract.

```java
public class ConcreteChild
                              extends AbstractParent {
   @Override
   public void aMethod() {
      System.out.println("Hi!");
   }
}
```

A Closer Look at Abstract Classes



```java
public abstract class Thermometer {
  private double degrees;

  public Thermometer(double degrees) {
    this.degrees = degrees;
  }

  public double getTemperature() {
    return degrees;
  }

  public void setTemperature(
                          double degrees) {
     this.degrees = degrees;
  }

  public abstract double getFreezingPoint();

  public abstract double getBoilingPoint();

  public abstract char getScale();
}
```

A Closer Look at Abstract Classes

An ***abstract class*** must be declared with the `abstract` modifier.

Abstract classes may contain ***fields***...

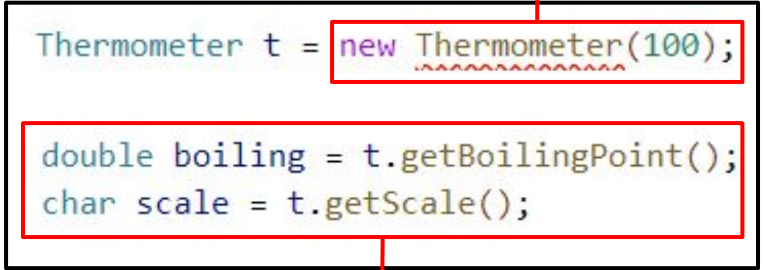
…***constructors***...

…and ***concrete methods***.

An abstract class may also define ***zero or more*** `abstract` methods.

```
public abstract class Thermometer {
  private double degrees;

  public Thermometer(double degrees) {
    this.degrees = degrees;
  }

  public double getTemperature() {
    return degrees;
  }

  public void setTemperature(
                          double degrees) {
     this.degrees = degrees;
  }

  public abstract double getFreezingPoint();

  public abstract double getBoilingPoint();

  public abstract char getScale();
}
```

An abstract class cannot be ***instantiated*** using `new` even if it defines a constructor. Why?

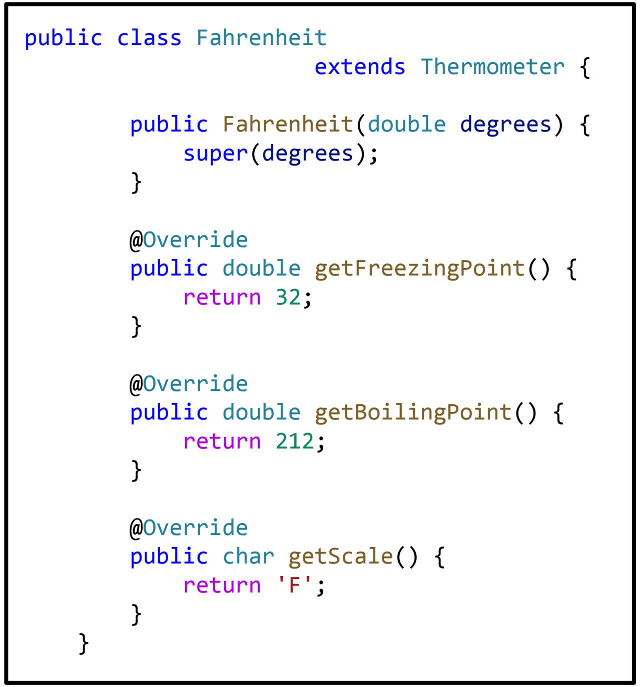


```
Thermometer t = new Thermometer(100);

double boiling = t.getBoilingPoint();
char scale = t.getScale();
```

If you could instantiate an abstract class, what would happen when you tried to call one of the abstract methods?

There is ***no code*** in those methods. What would Java execute? How would it know what to ***return***?

Subclassing an Abstract Class



```java
public class Fahrenheit
                      extends Thermometer {

        public Fahrenheit(double degrees) {
            super(degrees);
        }

        @Override
        public double getFreezingPoint() {
            return 32;
        }

        @Override
        public double getBoilingPoint() {
            return 212;
        }

        @Override
        public char getScale() {
            return 'F';
        }
    }
```

Subclassing an Abstract Class

A ***child class*** may subclass an abstract class using the `extends` keyword.

If the parent class defines any constructors, one of them must be called using `super`.

The child class ***must*** provide an implementation of each of the abstract methods defined by its parent class.

If the child class does not implement one or more of the abstract methods, it must also be declared `abstract`.

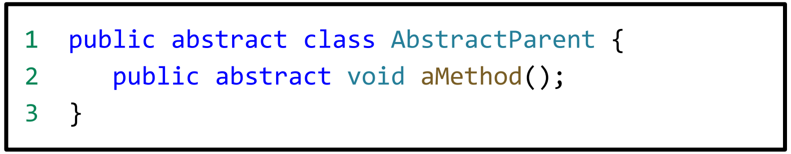
```java
public class Fahrenheit
                      extends Thermometer {

    public Fahrenheit(double degrees) {
        super(degrees);
    }

    @Override
    public double getFreezingPoint() {
        return 32;
    }

    @Override
    public double getBoilingPoint() {
        return 212;
    }

    @Override
    public char getScale() {
        return 'F';
    }
}
```

Abstract methods are used when there is no common implementation. Other child classes will implement the methods differently.

```java
public class Celsius extends Thermometer {

    public Celsius(double degrees) {
        super(degrees);
    }

    @Override
    public double getFreezingPoint() {
        return 0;
    }

    @Override
    public double getBoilingPoint() {
        return 100;
    }

    @Override
    public char getScale() {
        return 'C';
    }
}
```

3.18 An Abstract Goat

Abstract classes allow us to ***define*** behavior without ***implementing*** it. This seems like a perfect fit for some of the methods in the `Goat` class. Modify the `Goat` class so that it is `abstract` and convert any methods that do not have useful implementations into `abstract` methods.

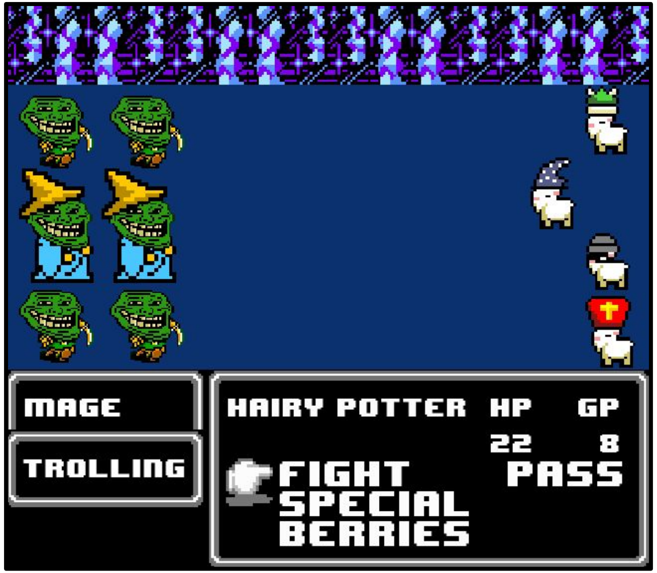


```java
public abstract class AbstractParent {
    public abstract void aMethod();
}
```

- Open the `Goat` class and convert it to an ***abstract class***.
  - Add the `abstract` modifier to the class declaration.
- Search through each method in the class and, if the method is not useful as implemented, convert it to an ***abstract method***.
  - Add the `abstract` modifier to the method declaration.
  - ***Delete*** the body of the method.
- What changes are necessary in any of the `Goat` child classes?
  - Why?
- ***Run*** your code to make sure that it still works.

Artwork by Leonid Afremov

The Trolls



Our Goats need Bad Guys to fight! Let's make some trolls.

In GvTV, trolls spawn and attack the party in waves. The number and type of trolls varies.

Type	Hit Points	Attack	Special
Trolling	38	**U Mad?** 15 (Physical Damage)	Takes +25% Magical Damage Regenerates 3%
Trollzord	64	**Flame War** 25 (Magical Damage)	Takes +25% Holy Damage Regenerates 5%

3.19 Another Simple Domain Analysis

We'll need new classes to represent each of the Trolls. Fill in the table below to identify the state and behavior needed for each of the new enemy classes.

Things to Note:

- A troll's **name** is the same as its class, e.g. "Trolling".
- Trolls **regenerate** health at the start of each round.
- When a troll's hit points reach 0, the troll is **vanquished** permanently.

Class	State	Behavior
 Trolling		
 Trollzord		

3.19 Another Simple Domain Analysis

We'll need new classes to represent each of the Trolls. Fill in the table below to identify the state and behavior needed for each of the new enemy classes.

Things to Note:

- A troll's **name** is the same as its class, e.g. "Trolling".
- Trolls **regenerate** health at the start of each round.
- When a troll's hit points reach 0, the troll is **vanquished** permanently.

Class	State	Behavior
 Trolling	NAME ("TROLLING") MAX HP CURRENT HP	GETTERS/SETTERS IS VANQUISHED? REGENERATE ATTACK TAKE DAMAGE
 Trollzord	NAME ("TROLLZORD") MAX HP CURRENT HP	GETTERS/SETTERS IS VANQUISHED? REGENERATE ATTACK TAKE DAMAGE

Software Design

Many novice programmers (and some who are experienced) use ***"clean sheet coding"*** in which they simply start writing code in an empty file with ***little thought*** put into an up front design.

This kind of software development is perfectly fine for ***throwaway code***; rapid prototypes or solutions to toy problems. In those circumstances, spending a lot of time on a design ***probably*** isn't worth it.

But any code that you plan to spend more than a few minutes on is worth designing ***up front***. This can (and will) save you a ***lot*** of time, especially with OO programs that include lots of little classes.

The good news is that, over time and with practice, you will get ***better*** and ***faster*** at designing software.

And you will learn about ***design patterns*** that can act like shortcuts to a design for common problems.

- Domain analysis is often the first step in ***software design***, and we have now performed ***simple domain analyses*** a number of times.
- In the past, we forged ahead and immediately began writing classes based on this initial analysis, but we now know that can lead to lots of revision.
  - For example, we ended up doing a lot of copying, pasting, and deleting code while implementing the first few `Goat` classes.
- Before tackling the Trolls, let's learn from our previous mistakes and try to anticipate a more elegant design before we dive into code.
  - A ***little more*** time spent on an initial design can save a ***lot more*** time spent writing, revising, and rewriting code later.
- Before moving forward, let's try to:
  - Identify potential ***parent classes*** based on common attributes among the Trolls.
  - Identify possible ***abstract methods*** where there is a method that every Troll needs, but there is no common implementation of the method.
- How will this extra effort ***now*** save us time and effort ***later***?

3.20 Identifying Common Attributes

Let's iterate on our domain analysis again and try to identify common attributes between the two different types of Trolls. This will help us make the decision about whether or not we need a common parent.

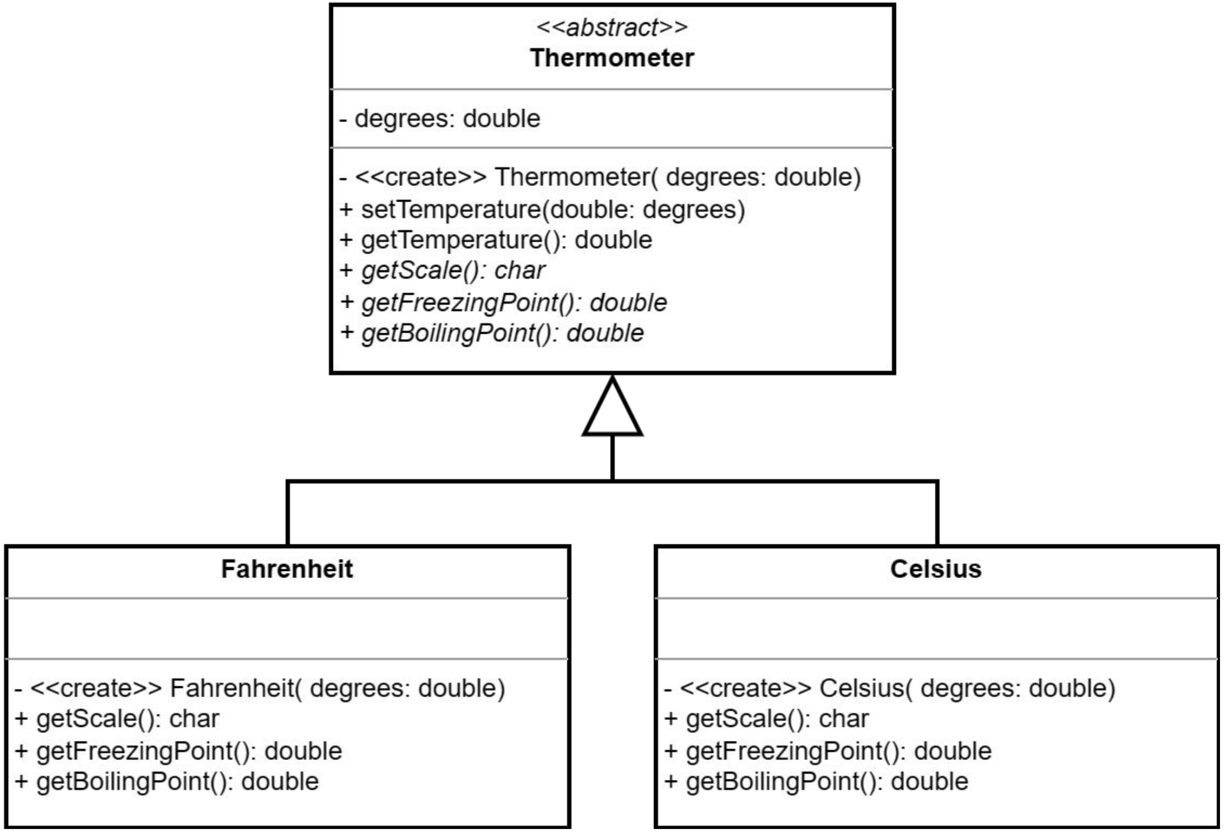
	Common to Both	Unique to Trollzord	Unique to Trolling
State			
Behavior			

3.20 Identifying Common Attributes

Let's iterate on our domain analysis again and try to identify common attributes between the two different types of Trolls. This will help us make the decision about whether or not we need a common parent.

	Common to Both	Unique to Trollzord	Unique to Trolling
State	NAME MAX HP CURRENT HP REGEN %	?	?
Behavior	CONSTRUCTOR? GETTERS/SETTERS REGENERATE? VANQUISHED?	CONSTRUCTOR ATTACK REGENERATE? TAKE DAMAGE	CONSTRUCTOR ATTACK REGENERATE? TAKE DAMAGE

Abstract Classes in UML



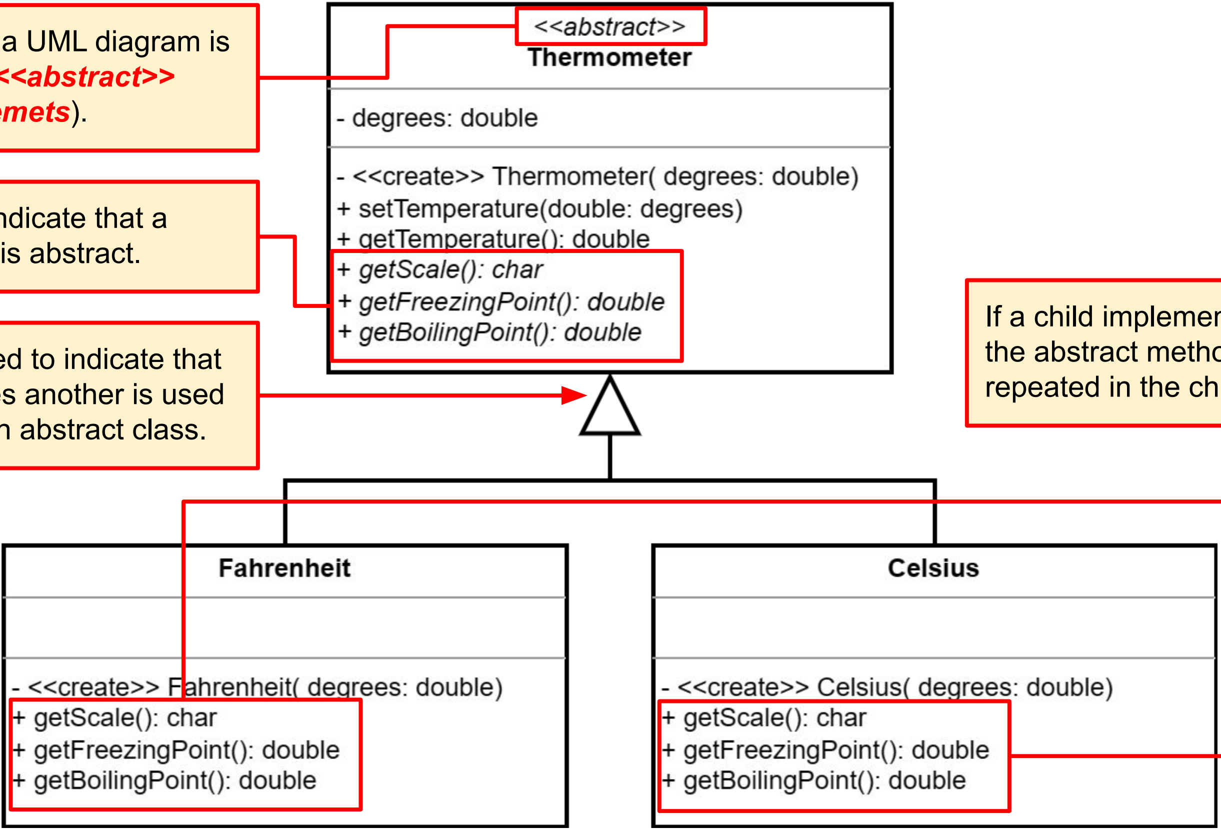
<<abstract>>
Thermometer
- degrees: double
- <<create>> Thermometer(degrees: double)
+ setTemperature(double: degrees)
+ getTemperature(): double
+ *getScale(): char*
+ *getFreezingPoint(): double*
+ *getBoilingPoint(): double*
Fahrenheit
- <<create>> Fahrenheit(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double
Celsius
- <<create>> Celsius(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double

Abstract Classes in UML

An abstract class in a UML diagram is indicated using the ***<<abstract>>*** annotation (in ***guillemets***).

Italics are used to indicate that a method in the class is abstract.

The same arrow used to indicate that one class subclasses another is used when subclassing an abstract class.

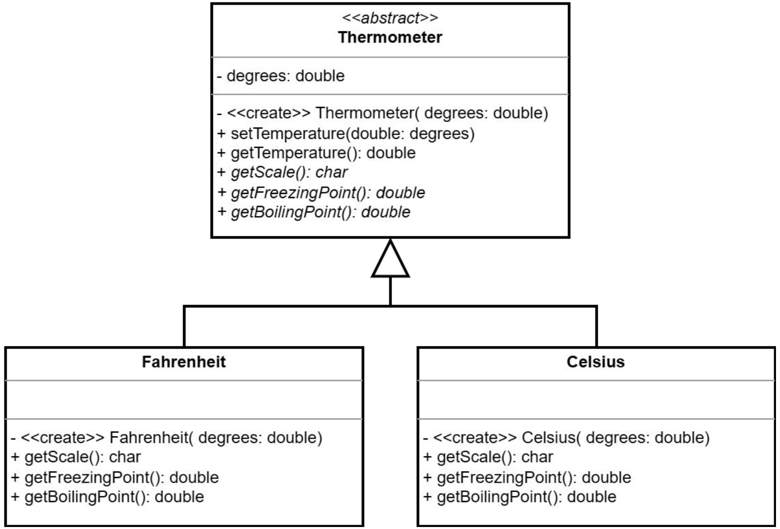


<<abstract>>
Thermometer
- degrees: double
- <<create>> Thermometer(degrees: double)
+ setTemperature(double: degrees)
+ getTemperature(): double
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double
Fahrenheit
- <<create>> Fahrenheit(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double
Celsius
- <<create>> Celsius(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double

If a child implements one or more of the abstract methods, they are repeated in the child class.

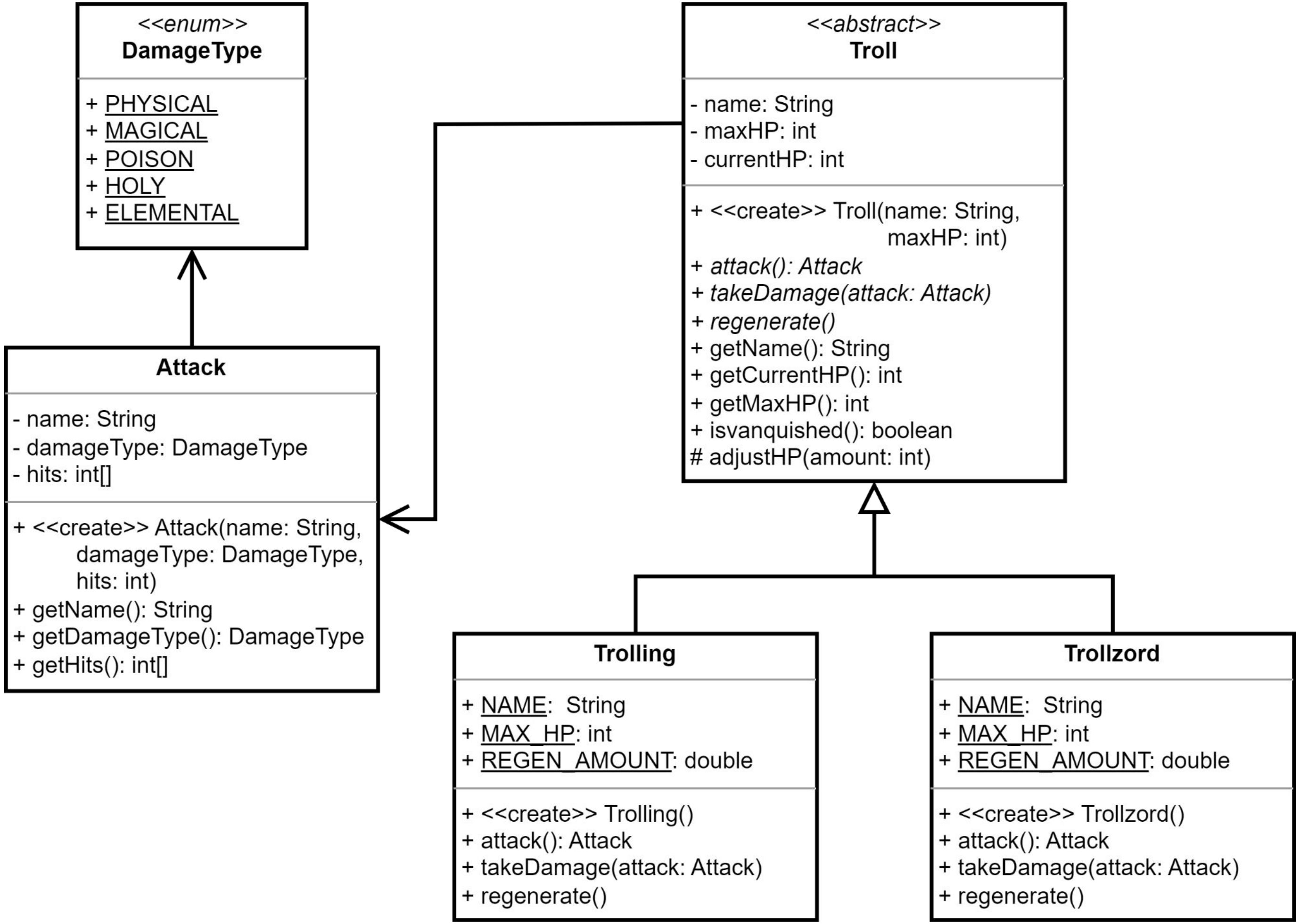
3.21 Diagramming Trolls

It is considered best practice to attempt to design your software before you start implementing it. Putting thought into the different classes that you will need, and how they are related to or associated with each other can help speed up development and save time in the long run.



<<abstract>>
Thermometer
- degrees: double
- <<create>> Thermometer(degrees: double)
+ setTemperature(double: degrees)
+ getTemperature(): double
+ *getScale(): char*
+ *getFreezingPoint(): double*
+ *getBoilingPoint(): double*
Fahrenheit
- <<create>> Fahrenheit(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double
Celsius
- <<create>> Celsius(degrees: double)
+ getScale(): char
+ getFreezingPoint(): double
+ getBoilingPoint(): double

- Using the drawing tool of your choice, create a UML class diagram representing the Troll classes that you are about to implement.
- You will need an ***abstract class*** to represent the common Troll state and behavior.
  - Which methods will you be able to implement?
  - Which should be abstract?
- You will also need classes for the *Trolling* and *Trollzord* classes.
- ***Challenge***: add the `DamageType` enum and the `Attack` class to your diagram.
  - Be sure to connect them to the other class(es) with appropriate UML relationships.



<<enum>>
DamageType

+ PHYSICAL
+ MAGICAL
+ POISON
+ HOLY
+ ELEMENTAL

<<abstract>>
Troll

- name: String
- maxHP: int
- currentHP: int

+ <<create>> Troll(name: String,
                 maxHP: int)
+ *attack(): Attack*
+ *takeDamage(attack: Attack)*
+ *regenerate()*
+ getName(): String
+ getCurrentHP(): int
+ getMaxHP(): int
+ isvanquished(): boolean
# adjustHP(amount: int)

Attack

- name: String
- damageType: DamageType
- hits: int[]

+ <<create>> Attack(name: String,
      damageType: DamageType,
      hits: int)
+ getName(): String
+ getDamageType(): DamageType
+ getHits(): int[]

Trolling

+ NAME: String
+ MAX_HP: int
+ REGEN_AMOUNT: double

+ <<create>> Trolling()
+ attack(): Attack
+ takeDamage(attack: Attack)
+ regenerate()

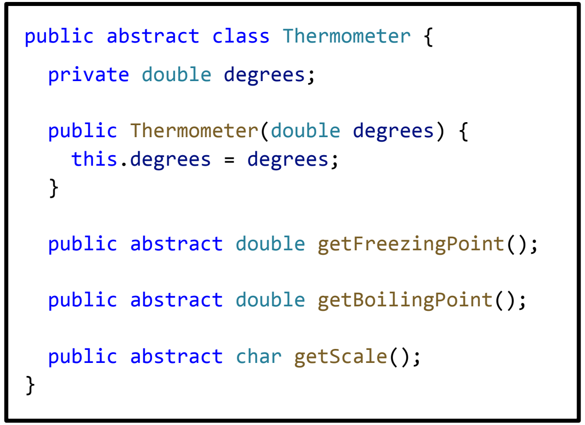
Trollzord

+ NAME: String
+ MAX_HP: int
+ REGEN_AMOUNT: double

+ <<create>> Trollzord()
+ attack(): Attack
+ takeDamage(attack: Attack)
+ regenerate()

3.22 A Common Troll

You now have enough of a design put together to begin implementing the common `Troll` class. Use your UML diagram as a guide to create the class now.

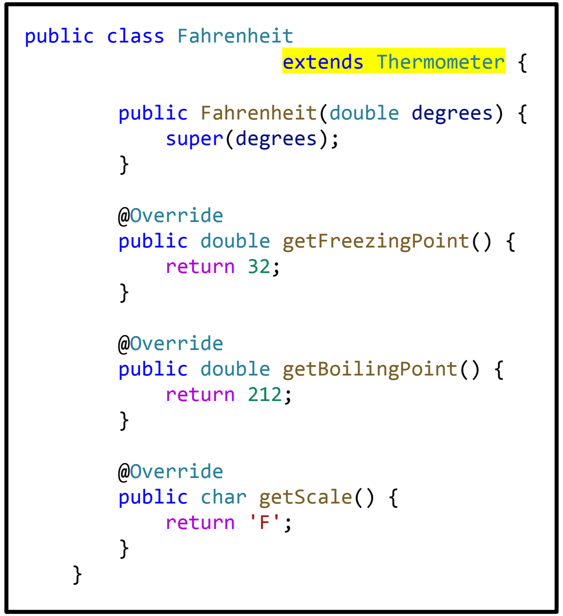


```java
public abstract class Thermometer {
  private double degrees;

  public Thermometer(double degrees) {
    this.degrees = degrees;
  }

  public abstract double getFreezingPoint();

  public abstract double getBoilingPoint();

  public abstract char getScale();
}
```

- Use your UML class diagram to guide you in creating a new Java class named "`Troll`".
  - Don't forget to use the `abstract` modifier in the class declaration.
  - Add any common state as ***fields***.
  - Add an appropriate ***constructor***.
  - Implement any common ***methods***.
  - Be sure to define any `abstract` ***methods*** that are needed by all Trolls but do not have a common implementation.

3.23 The Trollzord

Now that you have a common Troll class that provides common implementations of concrete methods and defines common behavior without implementing it, you're ready to create your first subclass: the Trollzord!

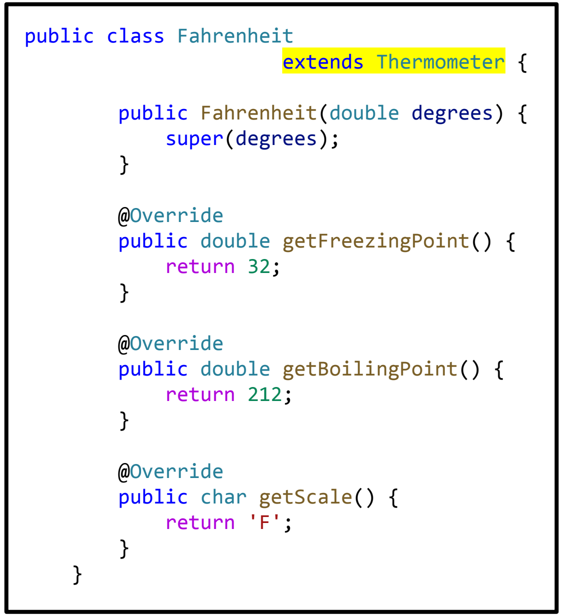


```
public class Fahrenheit
                                extends Thermometer {

        public Fahrenheit(double degrees) {
            super(degrees);
        }

        @Override
        public double getFreezingPoint() {
            return 32;
        }

        @Override
        public double getBoilingPoint() {
            return 212;
        }

        @Override
        public char getScale() {
            return 'F';
        }
    }
```

- Create a new Java class named "`Trollzord`". Remember that a `Trollzord`:
  - Is named `"Trollzord"`
  - Has ***64 hit points***
  - Attacks with **Flame War** which does ***25 magical damage***
  - Takes ***+25% holy damage***
  - Regenerates ***5% max health*** at the start of a round

3.24 The Trolling

Now that you have a common Troll class that provides common implementations of concrete methods and defines common behavior without implementing it, you're ready to create another subclass: the Trolling!



```
public class Fahrenheit
                    extends Thermometer {

        public Fahrenheit(double degrees) {
            super(degrees);
        }

        @Override
        public double getFreezingPoint() {
            return 32;
        }

        @Override
        public double getBoilingPoint() {
            return 212;
        }

        @Override
        public char getScale() {
            return 'F';
        }
    }
```

- Create a new Java class named "`Trolling`". Remember that a `Trolling`:
  - Is named `"Trolling"`
  - Has ***38 hit points***
  - Attacks with ***U Mad*** which does ***25 physical damage***
  - Takes ***+25% magical damage***
  - Regenerates ***3% max health*** at the start of a round

An Expression Parser

Most people are more familiar with ***infix notation***, using which the operator is situated between operands, e.g. ***2 + 2***.

However, parsing such expressions is very challenging. One technique is the ***Shunting- Yard Algorithm***.

Our expression parser will handle expressions written using ***prefix notation***, using which the operator ***precedes*** its operand(s), e.g. ***+ 2 2***.

Parsing expressions written using prefix notation is ***significantly*** easier, even though they can be a little harder for humans to read.

```
>>> + 2 * + 3 1 - 5 2
14
```

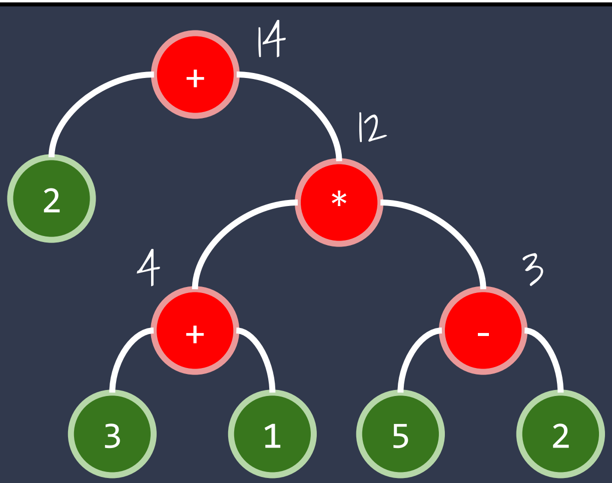
- Today we will be switching gears and implementing an ***expression parser***.
    - The user is prompted to type an arithmetic expression using ***prefix notation***.
    - The expression parser prints the results of evaluating the expression.
- An ***expression*** may be a ***constant*** or an ***operator*** performed on one or more ***operands***.
    - Each operand is another expression.
- The parser should support the following ***unary*** operations (meaning that there is ***one*** operand):
    - Increment (`++`)
    - Decrement (`--`)
- The parser should also support the following ***binary*** operations (meaning that there are ***two*** operands):
    - Addition (`+`)
    - Subtraction (`-`)
    - Multiplication (`*`)
    - Division (`/`)
    - Integer Division (`//`)
    - Exponent (`**`)
- The operand(s) for any supported operation may be ***any*** expression, including ***constants***, e.g.:
    - `* 4 5` (equivalent to `4 * 5`).
    - `+ * 3 2 - 4 5` (equivalent to `(3 * 2) + (4 - 5)`)

- Let's start by focusing on the ***expressions*** that our expression parser will create.
- When an ***expression*** is evaluated, the result is a single, floating point value.
- There are ***three*** basic types of expressions that our expression parser will support.
- A ***constant*** is a literal value, eg. `3.14159`.
  - When a ***constant expression*** is evaluated, it always just returns its value.
- A ***unary expression*** is an operator and exactly ***one*** other expression.
  - The other expression must be evaluated ***first***.
  - Then the operator is applied to the result.
  - For example ++ 5 is a unary expression that evaluates the constant expression 5, and then adds one.
- A ***binary expression*** is an operator and exactly ***two*** other expressions.
  - ***Both*** of the other expressions must be evaluated ***first***.
  - Then the operator is applied to both results.
  - For example + 4 5 is a binary expression that evaluates two constant expressions and then adds the results together.

Expressions

It can be helpful to visualize expressions as a ***tree***. For example, take the example expression shown previously.

```
+ 2 * + 3 1 - 5 2
```



Each ***binary expression*** in this tree must evaluate both of its expressions before applying the operator.

3.25 Expression Practice

Before we start implementing our expression parser, let's practice evaluating prefix expressions by hand.

Expression	Final Result
`7`	
`++ 5.6`	
`+ 2 3`	
`+ ++ 2 ++ 3`	
`+ 2 - 4 2`	
`* + 2 3 - 5 3`	
`+ - * 3 + 4 2 7 5`	

3.25 Expression Practice

Before we start implementing our expression parser, let's practice evaluating prefix expressions by hand.

Expression	Final Result
`7`	7
`++ 5.6`	6.6
`+ 2 3`	5
`+ ++ 2 ++ 3`	7
`+ 2 - 4 2`	4
`* + 2 3 - 5 3`	10
`+ - * 3 + 4 2 7 5`	16

3.26 Another Simple Domain Analysis

Let's start with a focused simple domain analysis and concentrate first on ***constants*** and the two ***unary expressions***: ***increment*** and ***decrement***.

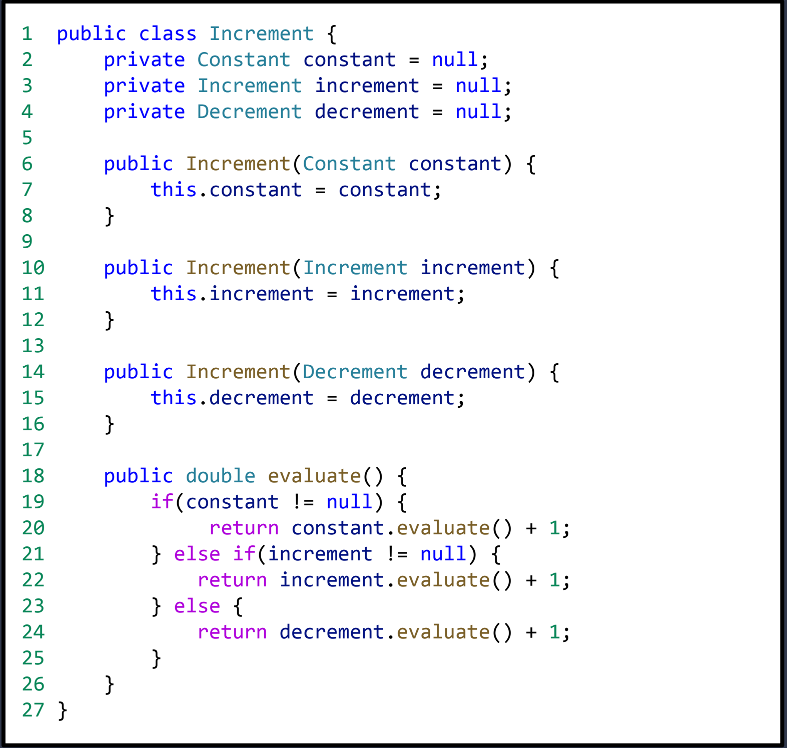
Class					
State					
Behavior					

3.26 Another Simple Domain Analysis

Let's start with a focused simple domain analysis and concentrate first on ***constants*** and the two ***unary expressions***: ***increment*** and ***decrement***.

Class	EXPRESSION	CONSTANT	UNARY EXPRESSION?	INCREMENT	DECREMENT
State	?	VALUE	EXPRESSION	EXPRESSION	EXPRESSION
Behavior	EVALUATE	CONSTRUCTOR EVALUATE	CONSTRUCTOR EVALUATE	CONSTRUCTOR EVALUATE	CONSTRUCTOR EVALUATE

A World Without Polymorphism



```java
public class Increment {
    private Constant constant = null;
    private Increment increment = null;
    private Decrement decrement = null;

    public Increment(Constant constant) {
        this.constant = constant;
    }

    public Increment(Increment increment) {
        this.increment = increment;
    }

    public Increment(Decrement decrement) {
        this.decrement = decrement;
    }

    public double evaluate() {
        if(constant != null) {
             return constant.evaluate() + 1;
        } else if(increment != null) {
            return increment.evaluate() + 1;
        } else {
            return decrement.evaluate() + 1;
        }
    }
}
```

Clearly this is not scalable. So let's make sure to use ***inheritance*** and ***polymorphism*** in our design.

- For many inexperienced ***object-oriented programmers*** thinking in terms of ***inheritance*** is not yet ***natural*** or ***automatic***.
- Nevertheless, it may seem clear that some kind of common parent class is needed to represent all ***expressions***.
  - This will enable us to use ***polymorphism*** in our ***unary*** and ***binary expression*** classes so that they will work with ***any*** kind of expression.
- However, let's take a moment and imagine a world where polymorphism doesn't exist. What would our expression classes look like then?
  - What would we need to do in order to make the ***increment*** expression work with ***any*** of our other expressions?
  - We would have no choice but to create fields for every type of expression, including constant, increment, and decrement.
- Our solution would only get worse as we added new expressions to the system, e.g. addition, exponent, etc.
  - ***Every*** expression class would need to have fields for ***every*** type of expression!

3.27 An Expression Class

Both constants and unary expressions are types of ***expressions*** that evaluate to a single value. This implies that they have some common behavior, even if the implementation of that behavior is going to be different. Create a common parent class to represent an expression.

```
public abstract class Animal {
   public abstract void speak();
}
```

Ultimately, we'd like our expression parser to be able to work with any kind of expression.

A common expression class will allow us to use ***polymorphism*** when implementing the parser.

- Create a new package named "`parser`" and use it for all of the code that you write for the "expression parser" problem.
- Create a new Java class name "`Expression`" in the `parser` package and define a method named `evaluate` that returns a `double`.
  - There is no possible common implementation of this method, and so it should probably be declared `abstract`.

3.28 A Constant Expression

A constant is the simplest kind of expression: it always evaluates to the same value. Create a new class that implements a Constant expression.

```
public abstract class Animal {
   public abstract void speak();
}
```

```
public class Dog extends Animal{
   private String name;
   public Dog(String name){
       this.name = name;
   }
   @Override
   public void speak(){
       System.out.println("Woof");
   }
}
```

- Create a new Java class named "`Constant`". When a constant expression is evaluated, it always returns its constant value.
    - Don't forget to ***extend*** your `Expression` class.
    - Declare a ***field*** to hold the constant value.
    - Implement a ***constructor*** to set the value.
    - ***Implement*** the `evaluate` method. Hint: it just returns the constant value!
- Define a `main` method with the appropriate signature, and test your new `Constant` class.

3.29 Increment & Decrement

Increment and decrement are both ***unary expressions***, meaning that each is created with ***one*** operand (another expression) that must be evaluated before the final result is computed.

++

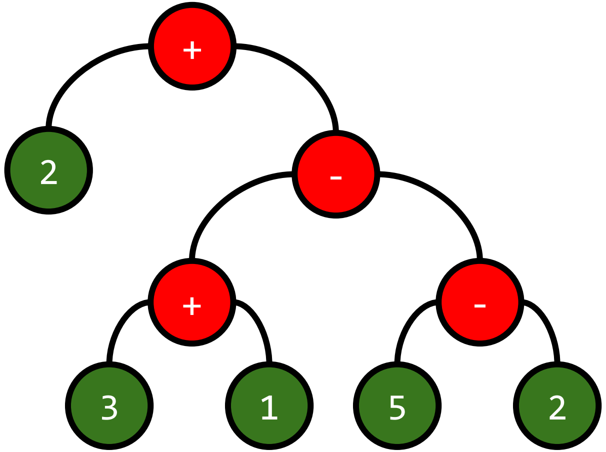
--

Unary expressions are created with some other expression. When evaluated, the unary expression first evaluates the other expression, then applies its operator.

- Create a new Java class named "`Increment`" and implement the ***increment*** expression.
    - Do not forget to ***extend*** your `Expression` class.
    - An increment expression is created with an ***operand*** that may be any other kind of expression.
    - When an increment is evaluated, it ***evaluates*** its operand expression and then ***adds 1***.
    - Add a `main` method to test your new `Increment` class.
- Repeat the above steps to create a new class that implements the ***decrement*** expression.

3.30 Addition & Subtraction

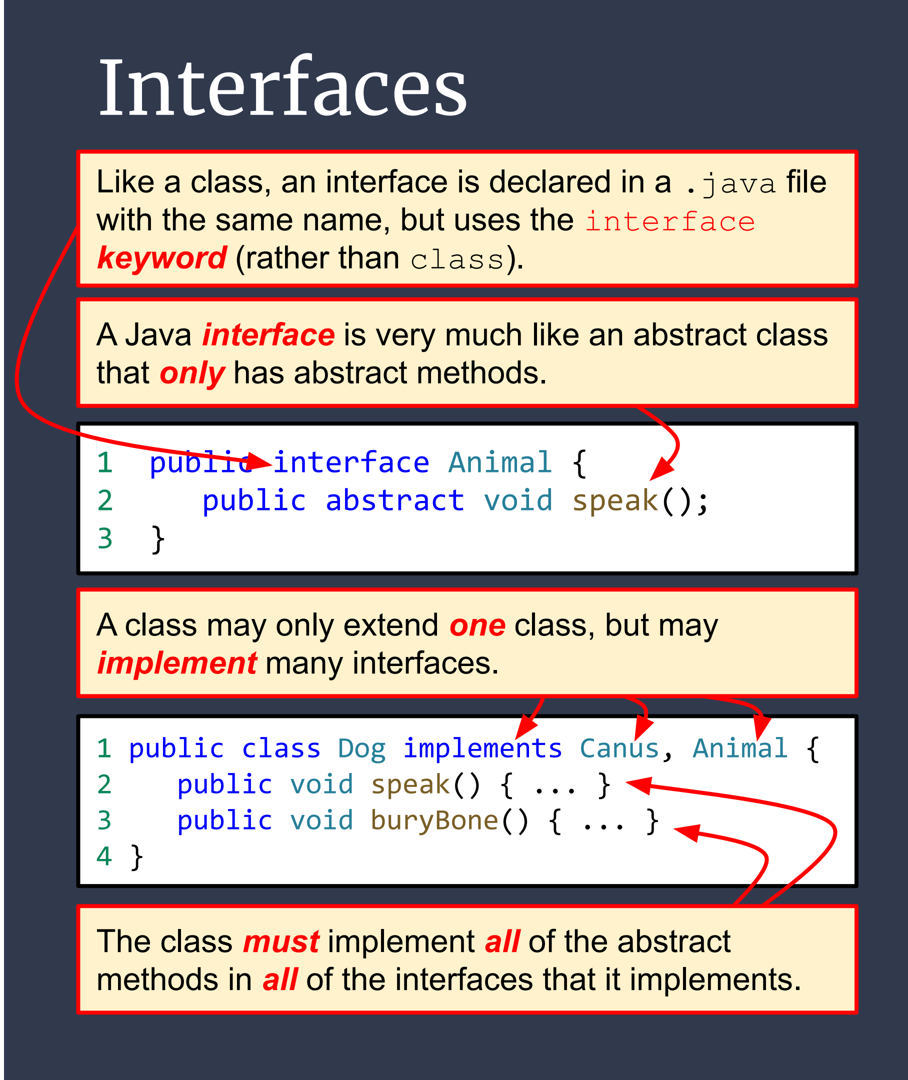
Addition and subtraction are ***binary expressions***. Each is created with ***two*** other expressions. ***Both*** expressions are evaluated before the operator is applied. Let's create classes to represent each of these binary expressions.



A binary expression must evaluate ***both*** of is expression before it can compute its result.

- Implement a new Java class in a file named "`Addition.java`" and implement the ***addition*** expression.
  - Do not forget to ***extend*** your `Expression` class.
  - An addition expression is created with ***two*** expressions that it uses as its operands.
  - When evaluated, the addition expression must first evaluate ***both*** of its operand expressions before ***adding*** the results together.
  - Add a `main` method to test your new `Addition` class.
- Repeat the above steps to create a new class that implements the ***subtraction*** expression.

- We now have ***five*** different expression classes: `Constant`, `Increment`, `Decrement`, `Addition`, and `Subtraction`.
- All five classes ***define*** the same behavior, but ***implement*** it differently: the `evaluate` method.
- As with the examples in the previous parts of this unit, we recognized the need for a parent class to ***define*** this behavior so that we could leverage ***polymorphism*** to use the different types of expressions ***interchangeably***.
  - But because there is no common implementation of the `evaluate` method that is shared by two or more classes, we declared the method `abstract`.
- But the `Expression` class has no other state or behavior; it's ***only*** method is abstract.
  - Unlike the `Goat` class, which had a ***mix*** of ***state***, and ***both*** concrete and abstract behavior.
- In Java, there is an alternative to abstract classes when there is no state and ***only abstract behavior***: the ***interface***.



Interfaces

Like a class, an interface is declared in a `.java` file with the same name, but uses the `interface` ***keyword*** (rather than `class`).

A Java ***interface*** is very much like an abstract class that ***only*** has abstract methods.

```java
public interface Animal {
    public abstract void speak();
}
```

A class may only extend ***one*** class, but may ***implement*** many interfaces.

```java
public class Dog implements Canus, Animal {
    public void speak() { ... }
    public void buryBone() { ... }
}
```

The class ***must*** implement ***all*** of the abstract methods in ***all*** of the interfaces that it implements.

Java Interfaces

When one class extends another, it is ***locked in*** to that relationship; it ***cannot*** extend any other class.

This means that ***polymorphism*** can't be used with any other Java class.

But a class may implement ***any number*** of interfaces (specified as a comma-separated list)...

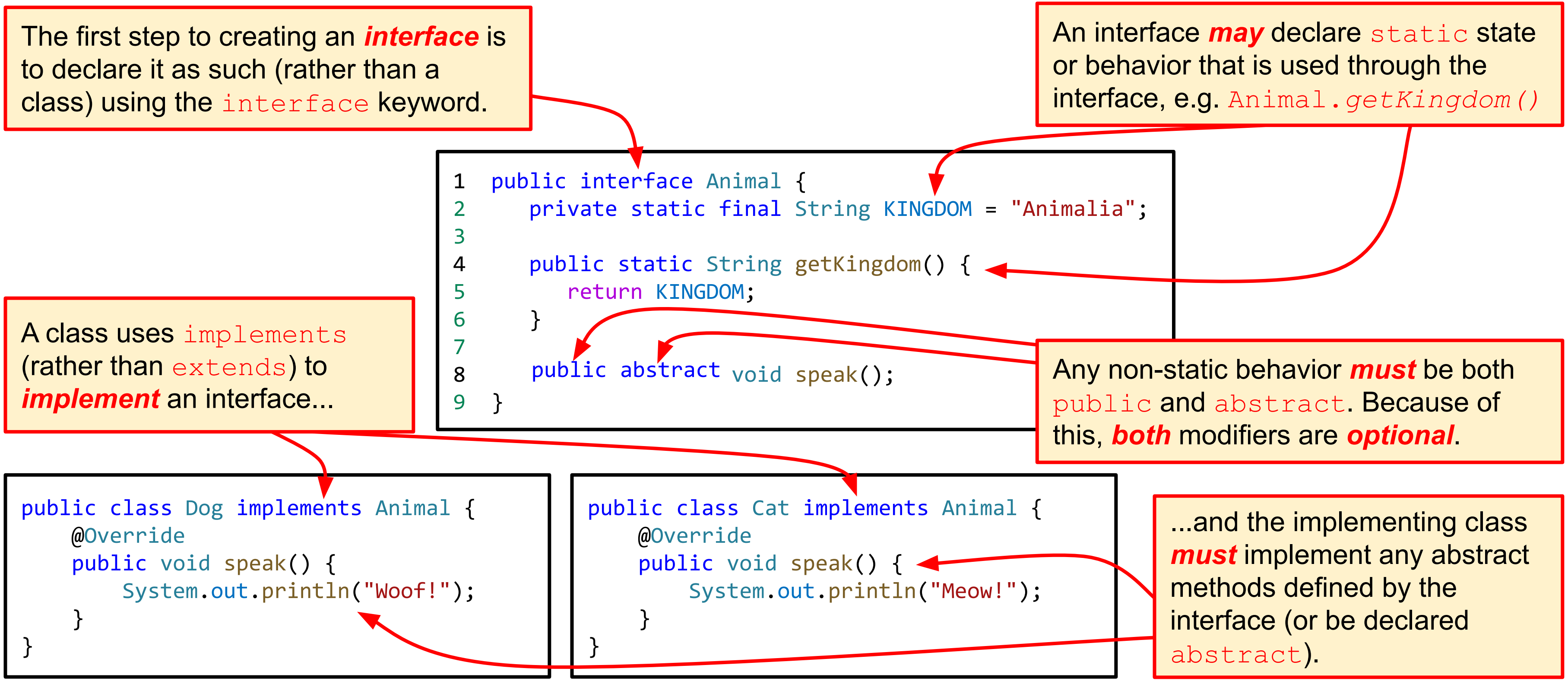
```
public class Implementor implements A, B, C, D {
    // ...
}
```

When a class implements an interface, it establishes an ***inheritance relationship*** with the interface, meaning that the class can be used in place of ***any*** interface that it implements.

Given the choice, using an interface instead of an abstract class makes your design ***much more flexible*** and dramatically improves the potential for ***polymorphism***.

- A Java ***interface*** is very much ***like*** an abstract class that contains ***only*** abstract methods.
  - An interface ***may*** include `static` state and behavior as well.
- An interface ***may not***:
  - Include any ***non-***`static` ***state***.
  - Include any ***concrete methods***.
  - ***Be instantiated*** using `new`.
- Creating an interface creates ***a new type***.
  - This means that an interface may be used as a variable, field, or parameter.
- A class that ***implements*** an interface establishes an inheritance relationship with the interface.
  - The class ***is a*** instance of the interface.
  - This means that any code written to use the interface will work with the class! ***Polymorphism!***
- Java only supports ***single inheritance*** with respect to classes.
  - A class may extend ***at most one*** other class.
  - If the class already extends some other class, it ***cannot*** extend a second class.
- But a class may implement ***any number*** of interfaces.
  - This means that, if an abstract class has ***only abstract methods***, it is better to use an interface instead.

A Closer Look at Interfaces



The first step to creating an ***interface*** is to declare it as such (rather than a class) using the `interface` keyword.

An interface ***may*** declare `static` state or behavior that is used through the interface, e.g. `Animal.getKingdom()`

```java
public interface Animal {
    private static final String KINGDOM = "Animalia";

    public static String getKingdom() {
        return KINGDOM;
    }

    public abstract void speak();
}
```

Any non-static behavior ***must*** be both `public` and `abstract`. Because of this, ***both*** modifiers are ***optional***.

A class uses `implements` (rather than `extends`) to ***implement*** an interface...

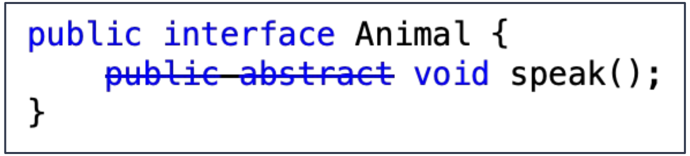
```java
public class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("Woof!");
    }
}
```

```java
public class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("Meow!");
    }
}
```

...and the implementing class ***must*** implement any abstract methods defined by the interface (or be declared `abstract`).

3.31 An Expression Interface

The `Expression` class defines only a single, abstract method. It does not have any fields, constructors, or concrete methods. There is no reason not to make it into an interface!



```
public interface Animal {
    ~~public abstract~~ void speak();
}
```

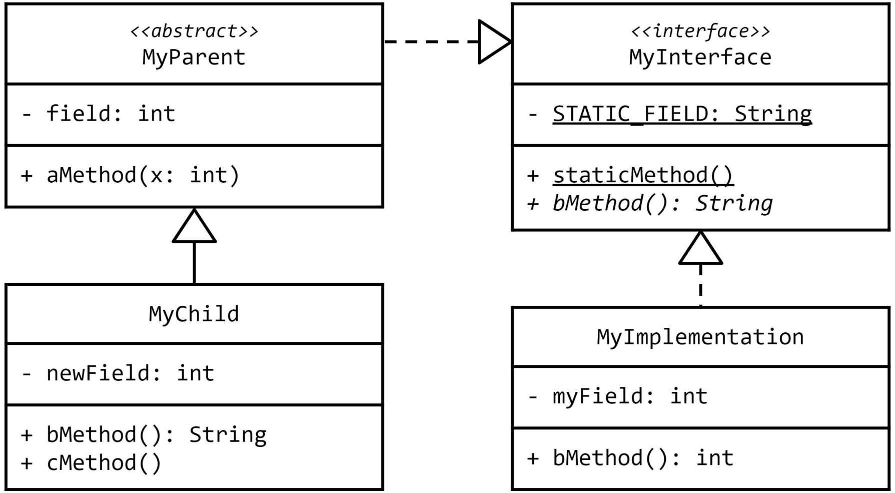
The `public` and `abstract` modifiers are optional in interfaces. It's a stylistic choice whether or not to include them.

- Open your `Expression` class and change it so that it is an ***interface*** rather than a class.
    - Change the "`class`" keyword to "`interface`".
    - Note that, because ***all*** methods in an interface ***must*** be both `public` and `abstract` that those modifiers are optional. You may remove them from your method declaration. This is a purely stylistic choice.
- You should notice that all of your other expression classes are now flagged by VSCode as having errors.
    - Update your concrete expressions so that they all ***implement*** your `Expression` interface.

`class` **VS.** `abstract class` **VS.** `interface`

Feature	Class	Abstract Class	Interface
May be instantiated using `new`.	✔	✘	✘
May include `static` state and behavior.	✔	✔	✔
May include non-`static` state.	✔	✔	✘
May define one or more `abstract` methods.	✘	✔	✔
Must implement inherited `abstract` methods.	✔	✘	✘
May implement concrete methods.	✔	✔	✘
May extend exactly one other class.	✔	✔	✘
May implement zero or more interfaces.	✔	✔	✘

UML
(with Interfaces)



<<abstract>>
MyParent

- field: int

+ aMethod(x: int)

<<interface>>
MyInterface

- STATIC_FIELD: String

+ staticMethod()
+ bMethod(): String

MyChild

- newField: int

+ bMethod(): String
+ cMethod()

MyImplementation

- myField: int

+ bMethod(): int

UML
(with Interfaces)

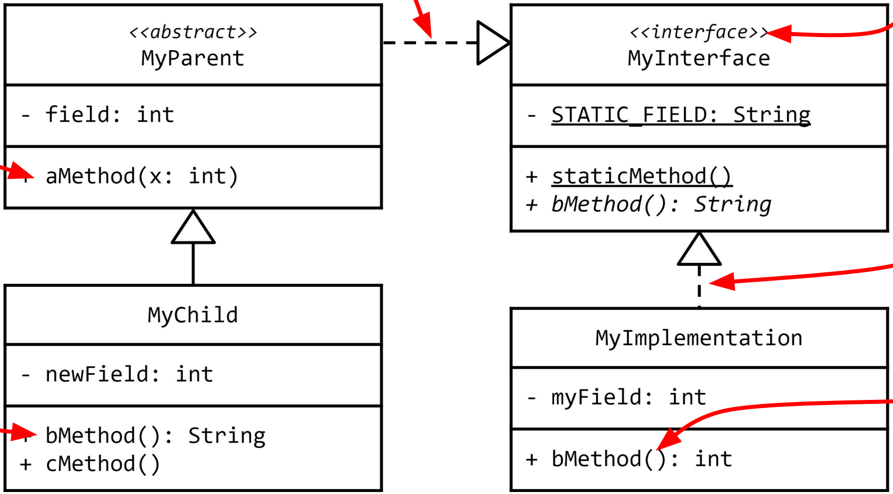
An abstract class ***does not*** need to implement any methods inherited from an interface...

...but the responsibility is then passed onto any ***concrete*** children that extend the abstract class.

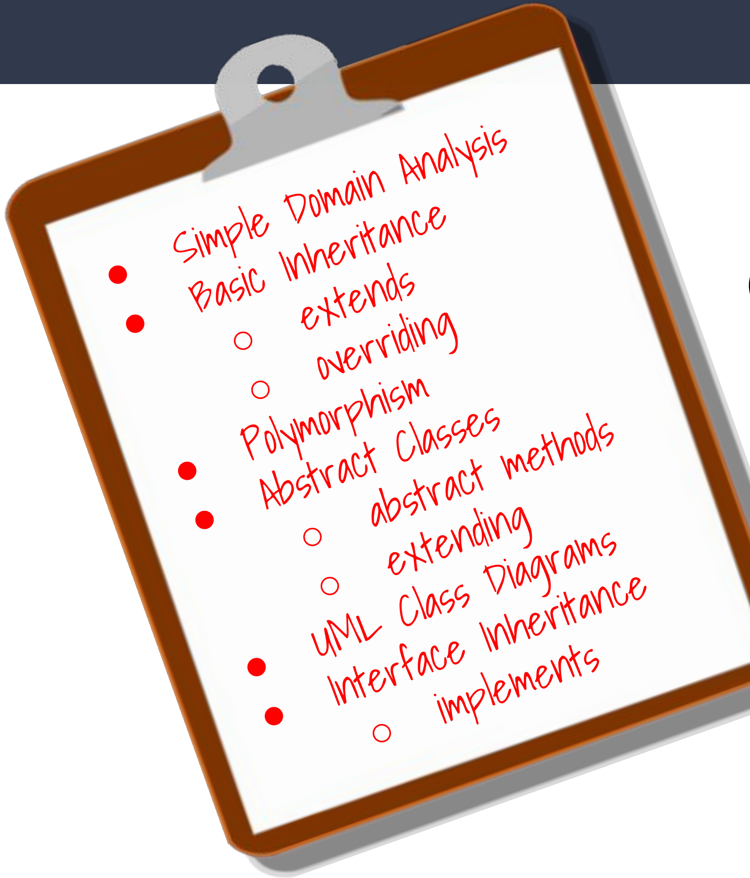
An ***interface*** is often marked with an *<<interface>>* tag above or below its name.

Implementation (also called ***"realization"***) is shown with a closed, unfilled arrowhead and a ***dashed line*** (---).

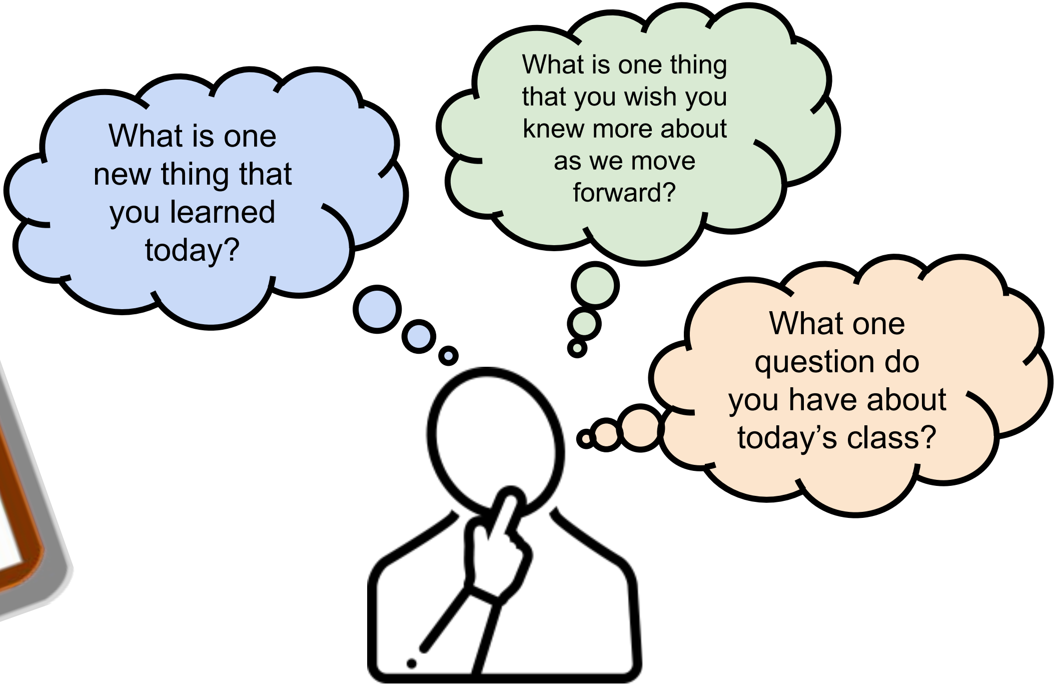
A class that ***implements*** an interface ***must*** implement the interface's methods (or be declared abstract).



Summary & Reflection



- Simple Domain Analysis
- Basic Inheritance
  - extends
  - overriding
- Polymorphism
- Abstract Classes
  - abstract methods
  - extending
- UML Class Diagrams
- Interface Inheritance
  - implements



What is one new thing that you learned today?

What is one thing that you wish you knew more about as we move forward?

What one question do you have about today's class?

Please answer the questions above in your notes for today.