

Pre-Class Work 3: Inheritance / Polymorphism

Inheritance [Background Reading]

□ Motivation

Imagine that you were the manager of a highly renowned restaurant ☐☐. You have your chefs and your servers, both of which work 40 hours a week, have an hourly rate of \$50, and have 10 vacation days. The main difference between the chef and the server is that the chef can cook food while the server can collect tips.

Consider the following code snippets below:

```
public class Chef {
    public int getHours() {
        return 40;
    }

    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public void cookFood(String order) {
        System.out.println(order + " order up!");
    }
}
```

```
public class Server {
    public int getHours() {
        return 40;
    }

    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public void collectTips(double amount) {
        System.out.printf("Ch-ching!! $%.2f\n", amount);
    }
}
```

```
}  
}
```

What do you notice between the two classes? The first three methods in both classes are the exact same! This introduces some concerns regarding code maintenance.

For example, if we wanted to increase the vacation days of all of workers to 15 days, you would have to change it in both the `Chef` and the `Server` class. While this is *okay*, you could imagine why this becomes difficult to maintain if we were to introduce more workers such as a cashier and dishwasher.

This is where we can introduce **inheritance** to help us.

Inheritance is a useful tool for organizing and structuring code in object-oriented programming. In particular, it allows you to create new classes that are based on existing ones. By creating a new class that inherits from an existing class, you can reuse the code from the existing class and add or modify the behavior as needed. This can save a lot of time and effort, as you don't have to write the same code again from scratch.

Inheritance also makes it easier to understand and maintain code, as it allows you to create a hierarchy of classes that reflect the relationships between different types of objects in your program. Finally, **inheritance** can be used to implement polymorphism, which allows a program to treat objects of different types in a uniform way (more information on this in the next reading).

□ Using `extends`

Inheritance is a way to form an *is-a* relationship between two classes. To form an *is-a* relationship, you need to include the keyword `extends`. If you have a class called `A` and you write a new class `public class B extends A`, you are saying that `B` is a **sub-class** of `A` and that `A` is a **super-class** of `B`. This relationship says that a `B` "is a" `A`, but is a more specialized version. For example you might have `Tiger extends Cat` since a tiger is a specialized version of a cat.

By default, a sub-class gets a copy of all the public methods defined in the super-class (we say the sub-class **inherits** the super-classes methods). The sub-class is able to **override** the methods to provide its own behaviors (This is a form of **polymorphism** which we will cover in the next reading).

A sub-class is able to access the methods of its super-class. If the sub-class wants to call `exampleMethod` on the super-class, it writes the code `super.exampleMethod()` ; .

A class can only extend up to one super-class.



If a class does not extend the class, it extends the class `Object` by default. This is why you need to **override** the `toString()` method, or else you'll inherit the one from the `Object` class.

To solidify this, let's take the code snippet above and form an **inheritance** relationship. First, let's

consider what `Chef` and `Server` have in common. As stated, they both share the first three methods in their class. More importantly, they are both employees for your restaurant. We can make a separate class called `Employee` which contain these shared methods:

```
public class Employee {
    public int getHours() {
        return 40;
    }

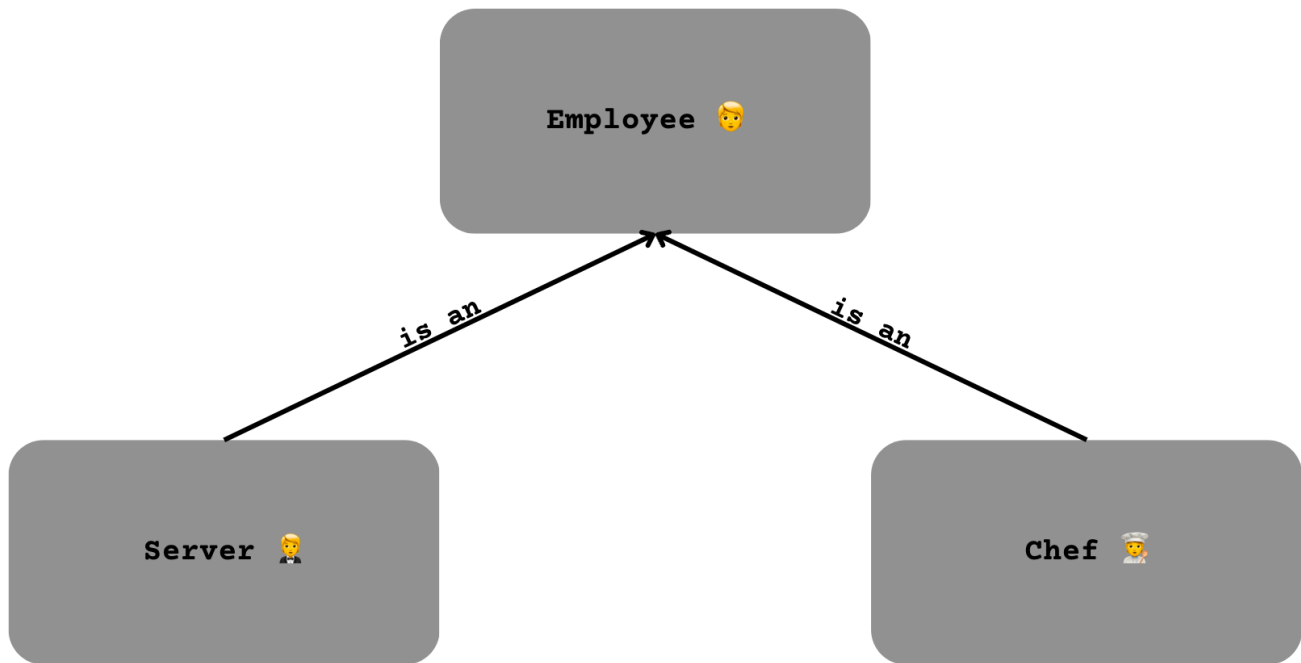
    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 10;
    }
}
```

```
public class Chef extends Employee {
    public void cookFood(String order) {
        System.out.println(order + " order up!");
    }
}
```

```
public class Server extends Employee {
    public void collectTips(double amount) {
        System.out.printf("Ch-ching!! $%.2f\n", amount);
    }
}
```

All `Employee`s work 40 hours a week, have an hourly rate of \$50, and have 10 vacation days. We can represent this **inheritance** relationship by drawing an **inheritance** tree:



Here, we are saying that `Chef` and `Server` are both `Employee`s, but they are specialized versions. A `Server` does everything an `Employee` does, but they can also collect tips. Similarly, a `Chef` does everything an `Employee` does, but they can cook food. Both `Server` and `Chef` *inherit* the methods of `Employee`. Now, if we wanted to give a `Server` and `Chef` more vacation days, we only need to make that modification in one class, the `Employee` class!

For example, if we wanted to give all `Employee`s 15 vacation days:

```
public class Employee {
    public int getHours() {
        return 40;
    }

    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 15; // Give all employees 15 vacation days.
    }
}
```

□ Using `super`

One thing to note is that when a subclass is created (i.e. when you use `extends`), it gains access to all the methods. However, constructors are not inherited by subclasses. The constructor of the superclass can still be called from within the subclass, but it is not automatically available to the

subclass like other class members. You can use the `super` keyword to refer to the superclass' constructor. For example:

```
public class Employee {
    private String employeeName;

    public Employee(String employeeName) {
        this.employeeName = employeeName;
    }

    public int getHours() {
        return 40;
    }

    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getEmployeeName() {
        return employeeName;
    }
}
```

```
public class Chef extends Employee {
    private int ordersFinished;

    public Chef(String employeeName) {
        super(employeeName);
        ordersFinished = 0;
    }

    public void cookFood(String order) {
        System.out.println(order + " order up!");
        ordersFinished++;
        System.out.println(super.getEmployeeName() + " has finished " + ordersFinished + " orders so
    }
}
```

```
public class Server extends Employee {
    private double totalTipsCollected;

    public Server(String employeeName) {
        super(employeeName);
        totalTipsCollected = 0.0;
    }

    public void collectTips(double amount) {
        System.out.printf("Ch-ching!! $%.2f\n", amount);
    }
}
```

```
totalTipsCollected += amount;
System.out.printf(super.getEmployeeName() + " has collected $%.2f worth of tips so far!\n", t
}
}
```

Notice here that `Chef` and `Server` both use the `super` keyword to refer to methods in the `Employee` class. The syntax is similar to using the `this` keyword but the difference is `super` is referring to the superclass and not *this* class.

One thing you might find interesting is the field `employeeName` in the `Employee` class. While fields are inherited, the subclass cannot directly access those fields. To read the value of a field, the subclass again needs to use the `super` keyword. This is a case where it is beneficial for the superclass to provide getter methods which in our case is `getEmployeeName()`.

Now, both `Chef` and `Server` can read the field `employeeName` by calling `super.getEmployeeName()`.

Now, what if we wanted to increase the hourly rate of a `Chef` but keep the hourly rate of the `Server` of a standard `Employee`? This is a case where we want to customize the behavior of a specific method so to do this, we can provide the `Chef` with their own `getHourlyRate()` method to **override** the `getHourlyRate()` in `Employee`.

In general, you can **override** the methods of the superclass in the subclass to customize the behavior of a method for a specific subclass. Let's say we wanted to increase the hourly rate of our `Chef` to \$500:

```
public class Employee {
    private String employeeName;

    public Employee(String employeeName) {
        this.employeeName = employeeName;
    }

    public int getHours() {
        return 40;
    }

    public double getHourlyRate() {
        return 50.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getEmployeeName() {
        return employeeName;
    }
}
```

```

public class Chef extends Employee {
    private int ordersFinished;


    public Chef(String employeeName) {
        super(employeeName);
        ordersFinished = 0;
    }

    public void cookFood(String order) {
        System.out.println(order + " order up!");
        ordersFinished++;
        System.out.println(super.getEmployeeName() + " has finished " + ordersFinished + " orders so
    }

    public double getHourlyRate() {
        return 500.0;
    }
}

```

Here, we are reimplementing an inherited method from a superclass. We are **overriding** the `getHourlyRate()` method in `Chef` to change the behavior of the inherited method.

 While optional, if you want to make this super explicit, you can also include the tag `@Override` above method:

```

@Override
public double getHourlyRate() {
    return 500.0;
}

```

Another approach we can do it modify the `Chef`'s hourly rate to scale with the hourly rate of a standard `Employee`. For example:

```

public class Chef extends Employee {
    private int ordersFinished;

    public Chef(String employeeName) {
        super(employeeName);
        ordersFinished = 0;
    }

    public void cookFood(String order) {
        System.out.println(order + " order up!");
        ordersFinished++;
        System.out.println(super.getEmployeeName() + " has finished " + ordersFinished + " orders so
    }

    public double getHourlyRate() {
        return super.getHourlyRate() * 10;
    }
}

```

In this code snippet, a `Chef`'s hourly rate is 10 times the rate of a standard `Employee`. Thus, when we change `getHourlyRate()` in the `Employee` class, it will also increase the hourly rate of a `Chef`! To visually see this, we have provided a client class in the next slide for you to execute.

□ Main Points

- **Inheritance** allows us to create new classes based on existing ones. We form an *is-a* relationship between two classes e.g. a `Server` *is-an* `Employee`.
- To allow for one class to inherit from another, we use the `extends` keyword when creating our class. `public class B extends A` indicates that class B is a **sub-class** of class A and that class A is a **super-class** of class B.
- **Sub-classes** inherit all the public methods from their **super-class**. They can use and/or **override** these methods.
 - You can put `@Override` above a method in the **sub-class** to explicitly indicate that you are overriding a method from the **super-class**.
 - This is a form of **polymorphism**, allowing different behavior for subclasses while still having a common interface.
 - **Constructors** are not inherited by **sub-classes**, but we can call a superclass constructor using `super()` in the **sub-class's** constructor to initialize fields inherited from the **super-class**.
- Within the **sub-class**, we can call methods from the **super-class** by using the `super()` keyword within a method or constructor.

Inheritance [Programming Example]

This slide contains the full code example from the previous reading. Feel free to look over the classes and try running the client!

Compiler Error vs. Runtime Error [Background Reading]

□ Declared Type and Actual Type

In the previous slide, whenever we would create an `Employee`, `Chef`, or `Server`, we made the declared type and object type (also known as actual type), the same:

```
Employee standardEmployee = new Employee("Frankie Blaze");
Server leadServer = new Server("Sam Taylor");
Chef headChef = new Chef("Julia Child");
```

However, we don't need the declared type and actual type to be the same. Recall that a `Chef` is an `Employee` and a `Server` is an `Employee`. Thus, we can declare we are creating `Employee`s but have the actual type be `Server` and `Chef`:

```
Employee standardEmployee = new Employee("Frankie Blaze");
Employee leadServer = new Server("Sam Taylor");
Employee headChef = new Chef("Julia Child");
```

This introduces some interesting scenarios. Let's investigate using `headChef`. What do you think would happen if we did:

```
headChef.getEmployeeName();
```

If you think this would still compile, you are correct! Now, consider the next line:

```
headChef.cookFood("Potatoes");
```

This line will throw an error, specifically a **compiler error**! You might be wondering why since in the `Chef` class we have a method `cookFood(String order)` which a `Chef` should be able to call! Let's revisit our variable initialization:

```
Employee headChef = new Chef("Julia Child");
```

This syntax should look familiar to you! The declared type (left-side of the "=") of a variable in Java determines the methods that can be called on that variable. If the compiler cannot find a method being called on a variable in the declared type or any of its superclasses, it will give an error.

For example, if you have a variable of type `List` (an interface) and you call the `add` method on it, the compiler will check if the `List` interface or any of its superclasses define the `add` method. If the `add` method is found, the code will compile.

At runtime, the actual type of the object that the variable refers to (also known as the "runtime type") will be used to determine which version of the method to execute.

For example, if the runtime type of the object is `ArrayList`, the `add` method from the `ArrayList` class will be executed. If the method is not found in the runtime type, the **inheritance** hierarchy will be searched for a superclass that defines the method. In this case, since the code compiled without any errors, it is guaranteed that the `add` method will be found at runtime.

If you wanted access to all methods of an `ArrayList`, you would want your declared type to be `ArrayList`:

```
List<Integer> list1 = new ArrayList<>(); // Methods in List interface
ArrayList<Integer> list2 = new ArrayList<>(); // All methods of ArrayList
```

Because the declared type of `headChef` is an `Employee`, the only methods we can execute are those of `Employee`! Thus, `headChef` can't call `cookFood(String order)` because that method does not exist in `Employee`. There are a couple of ways to go about fixing this. The first way is just to have the declared type be `Chef` which we did earlier:

```
Chef headChef = new Chef("Julia Child");
```

The second way is to use **type casting**:

□ Casting

In Java, **type casting** is the process of treating an object of one type as if it were an object of another type. This is done by using the type in parentheses followed by the name of the object being cast.

For example, consider the following code:

```
Employee headChef = new Chef("Julia Child");
((Chef) headChef).cookFood("Potatoes");
```

This will compile! Now, even though the declared type of `headChef` is `Employee`, we **type cast** `headChef` to be `Chef` which gives it access to the `cookFood(String order)` method! Note that **type casting** does not persist. When we cast `headChef` to `Chef`, that does not permanently change the `headChef` to become a `Chef`!

```
Employee headChef = new Chef("Julia Child");
((Chef) headChef).cookFood("Potatoes");
headChef.cookFood("Potatoes"); // THIS WILL CAUSE AN ERROR
```

Type casting can be useful when you need to call a method on an object that is not defined in its actual type, but is defined in a super class or interface. However, it can also be dangerous, as it can cause errors if the object being cast is not actually an instance of the type being cast to.

□ Compiler Error vs Runtime Error

When working with **type casting**, you will encounter two types of error: **compiler error** and **runtime error**.

First, **compiler errors**. A **compiler error** occurs when the code does not follow the rules of the Java language and cannot be compiled. This can happen when your cast type or declared type does not implement the method that a client is trying to execute.

For example:

```
Chef headChef = new Chef("Julia Child");
System.out.println(((Employee) headChef).getEmployeeName()); // OK
((Employee) headChef).cookFood("Potatoes"); // The cast type does not implement cookFood so this will
```

```
Employee headChef = new Chef("Julia Child");
headChef.cookFood("Potatoes"); // The declared type (Employee) does not implement cookFood so this will
```

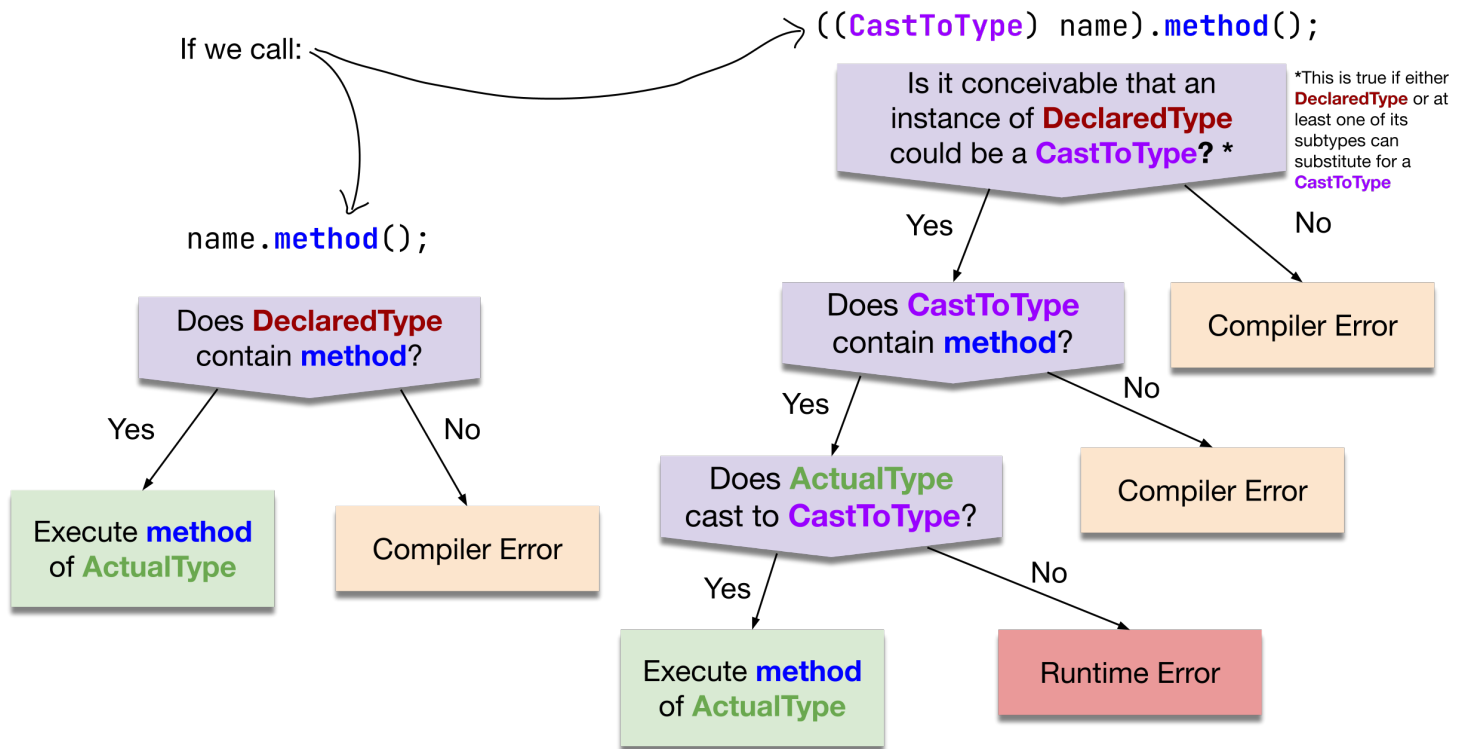
On the other hand, a **runtime error** occurs when the code compiles and runs correctly, but an error occurs while the program is executing. This can happen when you use type casting to cast a variable to a type that it is not actually an instance of.

For example:

```
// The following code will compile but it just won't execute.
Employee employeeOfTheMonth = new Employee("Frankie Blaze");
((Chef) employeeOfTheMonth).cookFood("Potatoes"); // An Employee is not a Chef! Thus, we can't cast
```

Learning to distinguish between a **compiler error** and **runtime error** takes practice. The following image below helps outline the thought process you should be taking in order to help you figure out whether something is a **compiler error** or **runtime error**:

With the following declaration and initialization:
`DeclaredType name = new ActualType();`



As we can see in the image, we will first define a few things.

We will declare the variable with the line `DeclaredType name = new ObjectType();`

Then we will call the method with the line `name.method();`

Then we will cast the object then call the method with the line `((CastToType)name).method();`

As we go through our flowchart, we will first ask whether we are casting.

- If we are casting, then we will ask if `CastToType` contains the method
 - If `CastToType` contains the method, then we will ask if we can cast `ObjectType` to `CastToType`
 - If we can cast `ObjectType` to `CastToType`, then we will execute the method of `ObjectType` □
 - If we *cannot* cast `ObjectType` to `CastToType`, then we will get a **runtime error** □
 - If `CastToType` does not contain the method, then we will get a **compiler error.** □
- If we are *not* casting, then we will ask if `DeclaredType` contains the method
 - If `DeclaredType` contains the method, then we will execute the method of `ObjectType` □
 - If `DeclaredType` does *not* contain the method, then we will get a **compiler error.** □

□ Main Points

- The **declared type** of a variable determines the methods you can call on that variable at *compile time*, while the **actual type** (*runtime type*) of the object determines which method implementation is executed at *runtime*.
 - We can use **declared types** to create hierarchies of objects e.g. a `Chef` is an `Employee`, so we can *declare* a `Chef` as an `Employee`.
- A **compiler error** happens when the code cannot be compiled and happens when the **declared type** or **cast type** doesn't implement the method being called.
- A **runtime error** happens when the code successfully compiles but there's an issue while executing. This usually happens when using type casting to cast an object to a type that it is not actually an instance of.
- **Type casting** is when we treat an object of one type as if it were an object of another type. We use the type in parentheses and then the object's name, such as `((CastToType)name).method()`.

Polymorphism [Background Reading]

Motivation

Polymorphism is often used in conjunction with **inheritance** so it's natural to include both topics in the same conversation. **Polymorphism** allows for the same interface to be used to invoke methods on objects of different types.

For example, consider a program that needs to perform some action on a collection of objects. Without **polymorphism**, the program would need to have a separate block of code for each type of object in the collection. You could imagine why this could become cumbersome once the number of object types grows.

With **polymorphism** however, the program can simply iterate over the collection and invoke a single method on each object, without needing to know the specific type of each object.

Simply put, **polymorphism** can reduce the amount of code that needs to be written and maintained, and make it easier to add new types of objects to the program.

Method Overriding

The first type of **polymorphism** which we briefly touched on when discussing **inheritance** is **method overriding**. This is when a subclass **overrides** a method of its superclass by providing a different implementation.

For example:

```
public class Animal {
    public void speak() {
        System.out.println("Random animal noise");
    }
}

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

Method Overloading

The second type of **polymorphism** is **method overloading**. This is when multiple methods of the same name have different method signatures. When a method is invoked, the Java compiler will determine the best fit method to invoke based on the parameters.

For example:

```
public class Animal {
    public void speak() {
        System.out.println("Random animal noise");
    }

    public void speak(int volume) {
        System.out.println("Making random animal noise at volume: " + volume);
    }

    public void speak(String noiseType) {
        System.out.println("Making noise of type: " + noiseType);
    }
}
```

Note that methods that are **overloaded** can still be **overridden**. For example:

```
public class Animal {
    public void speak() {
        System.out.println("Random animal noise");
    }

    public void speak(int volume) {
        System.out.println("Making random animal noise at volume: " + volume);
    }

    public void speak(String noiseType) {
        System.out.println("Making noise of type: " + noiseType);
    }
}

public class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Bark!");
    }

    @Override
    public void speak(int volume) {
        System.out.println("Barking at volume: " + volume);
    }

    // speak(String noiseType) gets inherited
}
```


□ Main Points

- **Polymorphism** allows for the same interface to be used to invoke methods on objects of different types so we can:
 - (1) reduce the amount of code that needs to be written and maintained, and
 - (2) make it easier to add new types of objects to the program
- **Method overriding** is when a subclass **overrides** a method of its superclass by providing a different implementation.
- **Method overloading** is when multiple methods of the same name have different method signatures.
- Methods that are **overloaded** can still be **overridden!**

Inheritance Mystery [Background Reading]

A good way to assess your understanding of **inheritance** and **polymorphism** is to work through inheritance mystery problems.

Consider the following code examples:

```
public class Husky extends Dog {
    public void method1() {
        System.out.print("Husky 1 ");
    }
    public String toString() {
        return "Husky " + super.toString();
    }
}

public class Animal {
    public void method1() {
        System.out.print("Animal 1 ");
    }
    public void method2() {
        System.out.print("Animal 2 ");
    }
    public String toString() {
        return "Animal";
    }
}

public class Pomeranian extends Dog {
    public void method1() {
        System.out.print("Pomeranian 1 ");
    }
}

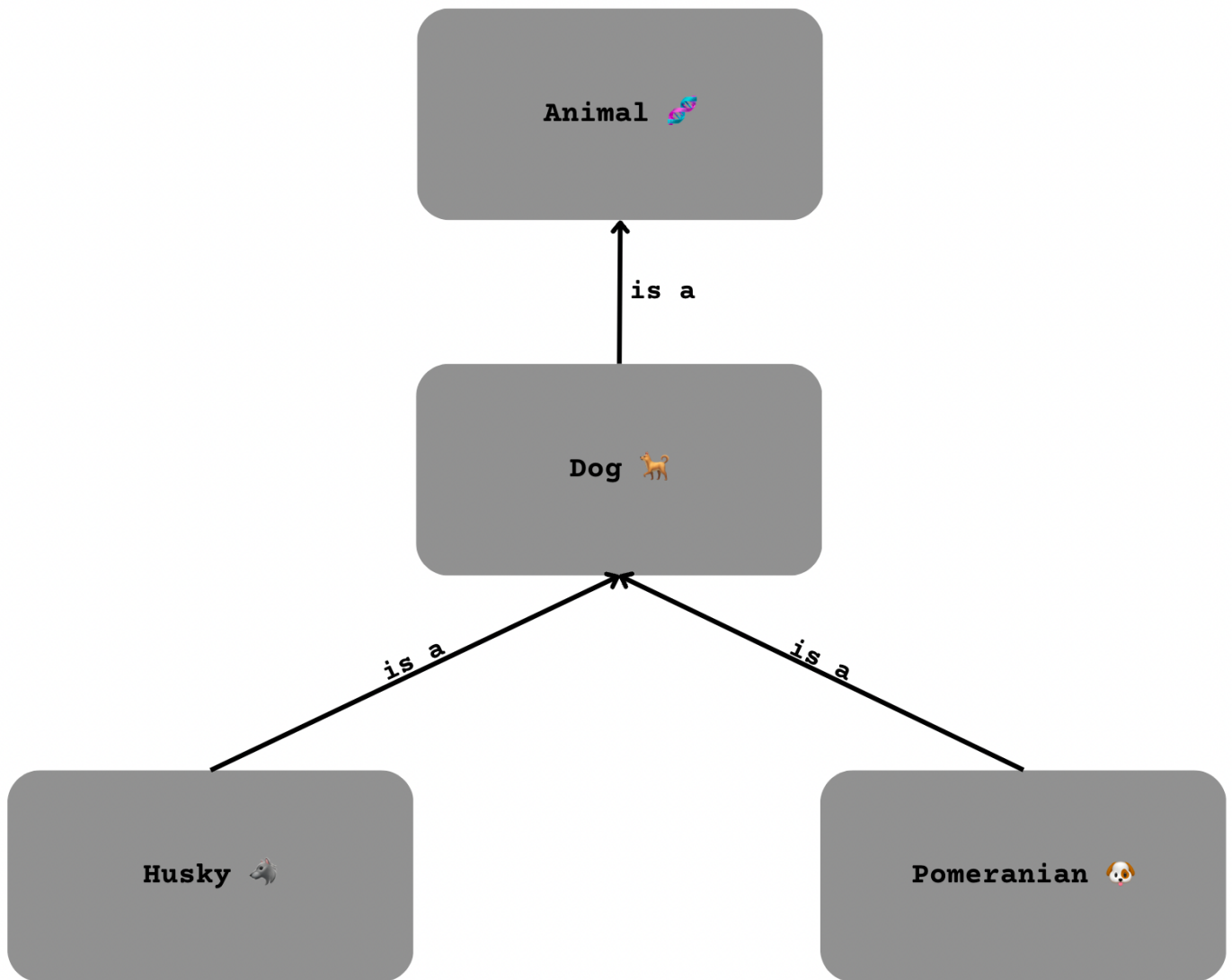
public class Dog extends Animal {
    public void method2() {
        method1();
        System.out.print("Dog 2 ");
    }
    public String toString() {
        return "Dog";
    }
}
```

Now, given these classes, think about what output of the from the following client class would be:

```
public class Client {
    public static void main(String[] args) {
        Animal[] elements = {new Animal(), new Dog(), new Husky(), new Pomeranian()};
        for (int i = 0; i < elements.length; i++) {
            elements[i].method1();
            System.out.println();
        }
    }
}
```

```
elements[i].method2();
System.out.println();
System.out.println(elements[i]);
System.out.println();
}
}
}
```

To help us solve this problem, let's first draw out an **inheritance** tree:



In this inheritance tree, we can see that **Husky** is a **Dog**, **Pomeranian** is a **Dog**, and **Dog** is an **Animal**.

Now, let's create a table to be a reference for what the output will be:

	method1	method2	toString
Animal			
Dog			
Husky			
Pomeranian			

□ Animal

First, let's consider the output of `Animal`. Since `Animal` is the base class there is no need look up in our **inheritance tree**. Thus, we can simply map the output of each method into their respective column:

	method1	method2	toString
Animal	Animal 1	Animal 2	Animal
Dog			
Husky			
Pomeranian			

□ Dog

Next, let's consider the output of `Dog`. Here, since `Dog` does not implement `method1()`, it **inherits** the method. Referencing our **inheritance tree**, we see that it will **inherit** `method1()` from `Animal`. Next, we see in `method2()` that the method calls `method1()` and then prints `Dog 2`. This is where things can get tricky. You might be tempted to say that `Dog`'s `method2` prints `Animal 1 Dog 2` which is correct. However, when you encounter a method call, you should avoid writing down the output in the table:

	method1	method2	toString
Animal	Animal 1	Animal 2	Animal
Dog	Animal 1	method1() Dog 2	Dog
Husky			
Pomeranian			

Notice that in the `method2` column that we do not immediately write down the output in the table. This is because method calls are **polymorphic** so we have to call `method1()` on the actual type of the object. This does not apply to super calls though as they are not polymorphic, which is why for a call like `super.toString()` call, we just look above and copy the output from that table entry.

If you are still unsure as to why we put `method1() Dog 2`, the next part should clarify it further.

□ Husky

Next, let's consider the output of `Husky`. Here, since `Husky` implements `method1()`, we can simply copy that output. The more interesting observation though is that `Husky` does not implement `method2()`. Thus, looking at our **inheritance tree**, `Husky` would **inherit** `method2()` from `Dog`.

Finally, in the `toString()` method, we make a call to `super.toString()` which means we can directly look above `Husky` in the **inheritance tree** to see what `toString()` method to reference. In this case, we will reference `Dog`'s `toString()` method.

	method1	method2	toString
Animal	Animal 1	Animal 2	Animal
Dog	Animal 1	method1() Dog 2	Dog
Husky	Husky 1	method1() Dog 2	Husky Dog
Pomeranian			

Notice that in the `method2` column that we directly copy whatever `Dog` had in its `method2` column. This is why we needed to put `method1() Dog 2` in `Dog`'s `method2` column instead of `Animal 1 Dog 2`. The actual output of `Husky`'s `method2` is `Husky 1 Dog 2`, not `Animal 1 Dog 2`!

□ Pomeranian

Finally, let's consider the output of `Pomeranian`. Since `Pomeranian` implements `method1()`, again, we can simply copy that output. Since `Pomeranian` doesn't implement `method2()`, that will get **inherited**. Referring the **inheritance tree**, we see that `Pomeranian` **inherits** `Dog`'s `method2()`.

Finally, since `Pomeranian` does not implement `toString()`, that method also gets inherited from `Dog`!

	method1	method2	toString
Animal	Animal 1	Animal 2	Animal
Dog	Animal 1	method1() Dog 2	Dog
Husky	Husky 1	method1() Dog 2	Husky Dog
Pomeranian	Pomeranian 1	method1() Dog 2	Dog

□ Final Output

Revisiting the client code:

```
public class Client {
    public static void main(String[] args) {
        Animal[] elements = {new Animal(), new Dog(), new Husky(), new Pomeranian()};
        for (int i = 0; i < elements.length; i++) {
            elements[i].method1();
            System.out.println();
            elements[i].method2();
            System.out.println();
            System.out.println(elements[i]);
            System.out.println();
        }
    }
}
```

We see now that the entire output would be as follows:

```
Animal 1
Animal 2
```


Animal

Animal 1

Animal 1 Dog 2

Dog

Husky 1

Husky 1 Dog 2

Husky Dog

Pomeranian 1

Pomeranian 1 Dog 2

Dog

□ Main Points

- **Inheritance** allows us to create a hierarchy of classes, where **sub-classes** inherit methods from their **super-classes**.
- **Sub-classes** inherit methods from their **super-classes**.
 - If a subclass does not explicitly define a method, it inherits that method from its **super-class**.
- **Sub-classes** can **override** methods inherited from **super-classes**.
 - The overridden method in the subclass takes precedence over the inherited one.
- **Polymorphism** allows objects of different types to be treated as objects of a common **super-class!**
- Drawing out an **inheritance tree** can help us track our polymorphism! :-)