

## Inheritance

Inheritance, encapsulation, abstraction, and polymorphism are the four fundamental concepts of object-oriented programming.

Inheritance enables new classes to receive—or *inherit*—the properties and methods of existing classes. You have seen that an object is a self-contained component that contains properties and methods needed to model a certain real world object. You have also seen that a **class is a blueprint** or template to build a specific type of object and that every object is built from an associated class definition.

Inheritance is a way to express a relationship between blueprints (classes). It's a way of saying: I want to build a new object that contains all of the data and behaviors of an existing, and instead of creating the new class from scratch, I want to reference the existing class in the new class' definition and simply indicate what's different.

Using the two basic concepts of inheritance

- subclassing (making a new class based on a previous one) and
- overriding (changing how a previous class works)

you can organize your objects into a hierarchy. Leveraging this hierarchy and overridden behaviors allows us to achieve polymorphism. Using inheritance to make this hierarchy often creates easier to understand code, but most importantly it allows you to reuse and organize code more effectively.

In object-oriented programming, inheritance enables new objects to take on the properties of existing objects. A class that is used as the basis for inheritance is called a **superclass or base class**. A class that inherits from a superclass is called a **subclass or derived class**. The terms *parent* and *child* are also acceptable terms you may encounter.

A child class inherits properties and methods from its parent similar to how human children inherit DNA code from their human parents and the hierarchies you can create in this fashion are similar to class hierarchies in the field of biological taxonomy, where the species in the lower levels of the hierarchy share characteristics with the species in the levels above. The items at the higher levels are more general (and may not even have concrete examples), while the items at the lower levels are more specific. Forks on the tree can happen at any level.



## “Is a” vs “Has a” relationship

The composition design pattern exhibited “has a” relationships

- An Employee **has a** hired date
- A Patient **has a** primary physician
- A physician **has a** list of specialities
- An engine **has a** starter
- A Character **has a** collection of items

The inheritance design pattern exhibits “is a” relationships. A subclass is a more specific instance of its superclass.

- A Doctor **is an** Employee
- A warlock **is a** character
- A sword **is a** weapon
- A lion **is a** mammal
- A rectangle **is a** polygon
- A square **is a** polygon

If the **is a** relationship does not exist between a subclass and superclass, you should not use inheritance. A sword **is a weapon**; so it is okay to write an ***Sword*** class that is a subclass of a ***Weapon*** class. As a contrast, a kitchen **has a** sink. It would not make sense to say a kitchen **is a** sink or that a sink **is a** kitchen. The **has a** relationship indicates composition rather than inheritance.

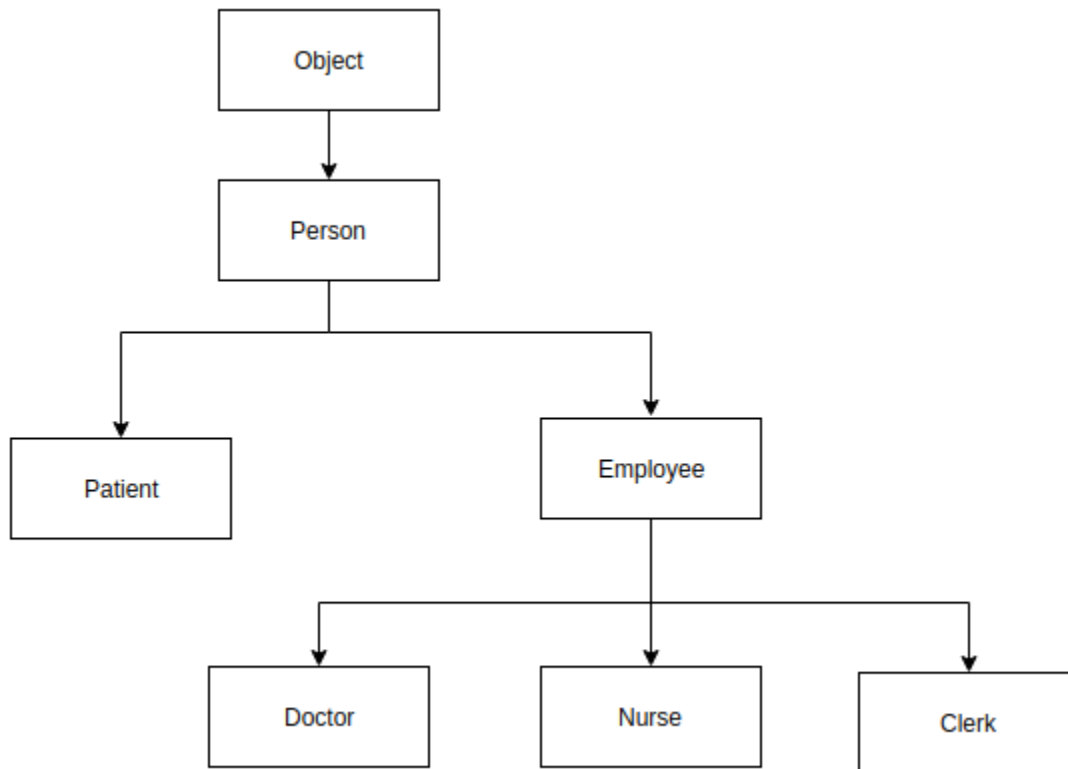
### Example: Doctor’s Appointments

You’ve been asked to design an application to track and process Doctor’s Appointments. This application will track

- Patients and their symptoms
- The clerk who checked them in
- The nurse who took their vitals
- The Doctor they visited
- The invoice for the visit

We want to leverage the composition and inheritance design patterns in this application and generalize and reuse as much code as possible.

With this in mind, let’s model the inheritance hierarchy for the classes Person, Patient, Employee, Clerk, Nurse and Doctor. We want to focus on grouping shared attributes into classes and then extending those classes into specific sub classes when necessary.



Notice how the hierarchy begins with the class `Object` as the ultimate ancestor. Every Java class extends `Object` even if this is not explicitly mentioned in the subclass. This association is provided by the Java compiler. You can see this relationship easily by creating a dummy class and calling `toString` on it.

Take a look at the example below. Notice on line 15 that I am able to invoke the `toString` method on instance `T` even though there is no mention of a `toString` definition in the `Toy` class. This is due to the inheritance mechanism providing this definition via extending `Object`. The inherited `toString` does not do anything appropriate for our class because we never specified how we wanted `toString` to **behave for our class** (foreshadowing!) All the inherited version of `toString` does is print the class type and the hash code.

**Module** `java.base`

**Package** `java.lang`

**Class** `Object`

`java.lang.Object`

---

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

**Since:**

1.0

```
1 public class Toy{
2
3     private String name;
4
5     public Toy(String name){
6         this.name = name;
7     }
8 }
9
10 class ToyDriver{
11
12     public static void main(String[] args){
13
14         Toy t = new Toy("Warlock Mini");
15         System.out.println(t.toString());
16
17     }
18 }
19
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac Toy.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java ToyDriver
Toy@6ff3c5b5
```

The version of toString that we have been writing is called an over ridden method. We will discuss this in more detail as we move through this module.

So, we have the top of the class hierarchy provided for us by the language. Let's define the root of our custom hierarchy by defining the Person class.

## Design Considerations

When designing a class hierarchy, you want to place the generic attributes and behaviors toward the top. These attributes will then be passed down through the hierarchy to the subclasses. In our situation, the most general attributes would be very general personal data: name, address, sex and date of birth. These attributes are shared by Doctors, Nurses, Patients and clerks. These should be defined as high up the hierarchy as possible . . . the Person class.

Also notice that we could use composition to handle the **Person has an address** association by defining Address as a class. An object like this could then be plugged in anytime an address needs to be tracked.

I have included a screen shot of a partial listing of this class. There are also toString and equals definitions. There are techniques in this equals method that will need some explanation, so I did not include it here.

```

1 public class Address {
2
3     private String street;
4     private String city;
5     private String county;
6     private String state;
7     private int zip;
8
9     public Address () {}
10
11     public Address(Address toCopy) {
12         this.street = toCopy.street;
13         this.city = toCopy.city;
14         this.county = toCopy.county;
15         this.state = toCopy.state;
16     }
17
18     public Address(String street, String city, String county, String state, int zip) {
19         this.street = street;
20         this.city = city;
21         this.county = county;
22         this.state = state;
23         this.zip = zip;
24     }
25
26     public String getStreet() {
27         return street;
28     }
29
30     public void setStreet(String street) {
31         this.street = street;
32     }
33
34     public String getCity() {
35         return city;
36     }

```

Now let's take a look at the listing for the Person class. Pay attention to the fields that could generate privacy leaks. These have been handled. If you are creating these classes as you work through this document be sure to write unit tests for the relevant features, particularly any fields dealing with object references and potential privacy leaks. You will have to do this from here on out.

```

1 public class Person {
2
3     private String firstName;
4     private String lastName;
5     private Date DOB;
6     private char sex;
7     private String phone;
8     private Address address;
9
10    public Person() {}
11
12    public Person(Person toCopy) {
13        this.firstName = toCopy.firstName;
14        this.lastName = toCopy.lastName;
15        this.DOB = new Date(toCopy.DOB);
16        this.sex = toCopy.sex;
17        this.phone = toCopy.phone;
18        this.address = new Address(address);
19    }
20
21    public Person(String firstName, String lastName, Date dOB, char sex, String phone, Address address) {
22        super();
23        this.firstName = firstName;
24        this.lastName = lastName;
25        this.DOB = new Date(dOB);
26        this.sex = sex;
27        this.phone = phone;
28        this.address = new Address(address);
29    }
30

```

```

51    public void setAddress(Address address) {
52        this.address = new Address(address);
53    }
54
55    public Date getDOB() {
56        return new Date(DOB);
57    }
58
59    public void setDOB(Date dOB) {
60        DOB = new Date(dOB);
61    }

```

## The “extends” keyword

Sub classes are considered an extension of the superclass. Defining a derived class from a base class is considered **extending the class**. We accomplish this with the keyword **extends**. Before we delve any further into the Doctor/Patient example I would like to create a small toy program that allows us to focus specifically on the details of extension.

Let's start this demonstration with the definition of **class A**. Take a look at the feature access modifiers. These play an important role in what is accessible in the sub-class.

```
public class A{
    private String name;
    public A(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    private void helper(){
        System.out.println("Do something helpful");
    }
    public String toString(){
        return "Class A --> Name is: " + name;
    }
} // end of A
```

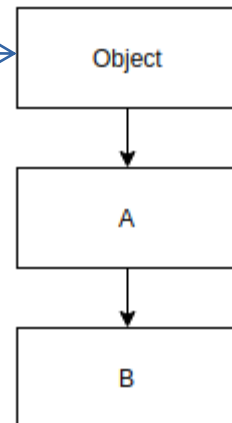
To extend a class in Java the superclass must reside somewhere in the CLASSPATH, which allows us to import; or be in the same directory as the subclass, which requires no import. For simplicity's sake I am defining these classes in the same directory.

We can define class B as an extension of A with the class header (this class is not public because I have it defined in the same source code file as class A and there can only be a single public class in a source file.)

```
class B extends A{
```

Now we have the following inheritance hierarchy

- A implicitly extends Object
- B explicitly extends A
- Instances of class B now have 3 associated data types {Object, A, B}



Here is the full listing of class B

```
class B extends A{
    private double weight;
    public B(String name, double weight){
        super(name);           // pass name to superclass A
        this.weight = weight;   // weight stays with "this"
    }
    public String toString(){
        // call the super class toString, concatenate to "this" toString
        return super.toString() + " Class B --> weight is: " + weight;
    }
} // end of B
```

### Important Concepts

- A is the super class (or base class); B is the sub class (or derived class)
- Private features in A stay private, and B cannot access those features directly. **They exist but cannot be accessed.**
- Public features in A can be access directly by B
- The constructor for class B accepts both **name and weight** attributes. There is no public setName method in A so we need to pass **name** through the A constructor. You refer to superclass constructors via the **super** keyword. Notice that you do not say **A(name)** it is **super(name)**
- The name variable gets passed up the hierarchy to super, while the weight variable gets assigned locally to **this.weight**
- The over ridden toString definition in B includes the result from the super class definition of toString. Notice the syntax **super.ToString()**. You have to prefix **super** to this method call because there are three definitions; one in Object, one in A and one in B. You cannot say **super.super.toString()** to access the Object's version of toString. Notice the output that B's toString gives.

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         B b = new B("warlock mini", 5);
43         System.out.println(b);
44     }
45 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java InheritanceDriver
Class A --> Name is: warlock mini Class B --> weight is: 5.0
```



Notice that we can invoke methods defined in class A as long as they are public

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         B b = new B("warlock mini", 5);
43         System.out.println(b);
44
45         String n = b.getName(); ← Invoke inherited method
46         System.out.println("Name is: " + n);
47     }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java InheritanceDriver
Class A --> Name is: warlock mini Class B --> weight is: 5.0
Name is: warlock mini
```

You cannot invoke private methods of the super class. These methods exist in memory but are private, and that restricts access even to subclasses.

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         B b = new B("warlock mini", 5);
43         System.out.println(b);
44
45         String n = b.getName();
46         System.out.println("Name is: " + n);
47
48         b.helper(); ← Method "helper" is private in super class
49     }
50 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

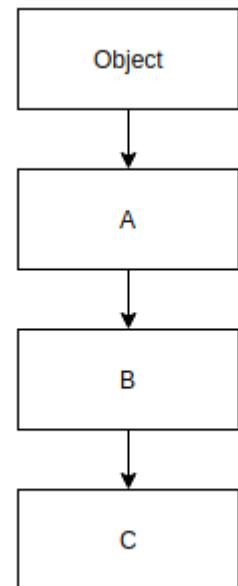
```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
A.java:48: error: cannot find symbol
    b.helper();
    ^
  symbol:   method helper()
  location: variable b of type B
1 error
```

Let's define class C as a subclass of class B

```
class C extends B{
    private String color;
    public C(String name, double weight, String color){
        super(name, weight);    // pass name and weight to superclass B
        this.color = color;    // color stays with "this"
    }
    public String toString(){
        return super.toString() + " Class C --> color is: " + color;
    }
} // end of C
```

This is our class hierarchy now

- C is an instance of B
- B is an instance of A
- Therefore C is an instance of A
- A, B and C are all instances of Object



In the code below, notice how the toString invocation on **c** includes the results from **B.toString** and **A.toString**.

We can easily call any public method that exists **above** “**this**” in the inheritance hierarchy. See line 46 below

**This arrangement does not work the other way around. A cannot call B methods, B cannot call C methods!!**

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         C c = new C("warlock mini", 5, "Red");
43         System.out.println(c);
44
45         // call an A method
46         String n = c.getName();
47         System.out.println("The name is: " + n);
48     }
49 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java InheritanceDriver
Class A --> Name is: warlock mini Class B --> weight is: 5.0 Class C --> color is: Red
The name is: warlock mini
```

## The instanceof Operator

Java provides the boolean operator **instanceof** to test an object's lineage. Use this operator whenever you are interested in where an object falls in an inheritance hierarchy. This operator will return true if the object in question is a direct instance or a subclass of the specified class. Here is an example with the A, B, C hierarchy.

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         C c = new C("warlock mini", 5, "Red");
43
44         System.out.println("Is C an instance of B? " + (c instanceof B));
45         System.out.println("Is C an instance of A? " + (c instanceof A));
46         System.out.println("Is C an instance of Object? " + (c instanceof Object));
47     }
48 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java InheritanceDriver
Is C an instance of B? true
Is C an instance of A? true
Is C an instance of Object? true
```

Notice how this returns true for each class above C in the hierarchy. If you want to know exactly which class defines an instance use the inherited method **getClass()**. This method is defined in the Object class. You can think of this method as telling you the specific constructor that was used when the object was instantiated . . . ie. which constructor was called with **new**.

```
40 class InheritanceDriver{
41     public static void main(String[] args){
42         C c = new C("warlock mini", 5, "Red");
43         B b = new B("wizard mini", 3);
44         A a = new A("halfling mini");
45
46         System.out.println("Is C an instance of B? " + (c instanceof B));
47         System.out.println("Is C an instance of A? " + (c instanceof A));
48         System.out.println("Is C an instance of Object? " + (c instanceof Object));
49     };
50     System.out.println("Was the B constructor called? " + (c.getClass() == b.getClass()));
51     System.out.println("Was the A constructor called? " + (c.getClass() == a.getClass()));
52 }
53 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ java InheritanceDriver
Is C an instance of B? true
Is C an instance of A? true
Is C an instance of Object? true
Was the B constructor called? false
Was the A constructor called? false
```

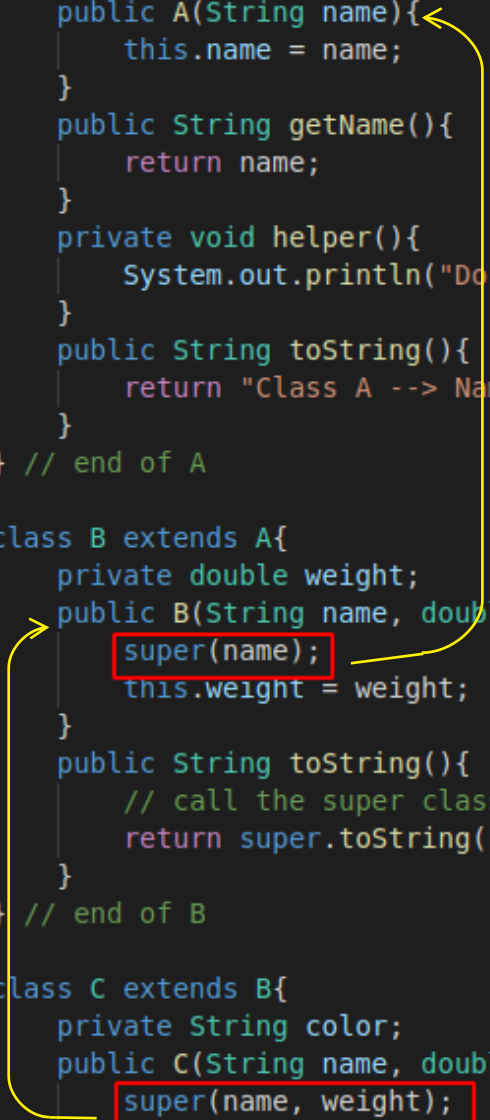
## Super Class Constructor Chaining

It is important to understand how this inheritance hierarchy is instantiated. If an instance of C is created using the following snippet

```
C c = new C("warlock mini", 5, "Red");
```

Then the constructors are called in this order:  $C \rightarrow B \rightarrow A \rightarrow \text{Object}$  We can trace our user defined constructor chain as there are explicit calls to the super class. Notice that there is no `super()` call in the constructor for A even though its super class is Object. When you omit this, the compiler will insert a no argument call to *super()*;

```
1  public class A{
2      private String name;
3      public A(String name){
4          this.name = name;
5      }
6      public String getName(){
7          return name;
8      }
9      private void helper(){
10         System.out.println("Do something helpful");
11     }
12     public String toString(){
13         return "Class A --> Name is: " + name;
14     }
15 } // end of A
16
17 class B extends A{
18     private double weight;
19     public B(String name, double weight){
20         super(name); // pass name to superclass A
21         this.weight = weight; // weight stays with "this"
22     }
23     public String toString(){
24         // call the super class toString, concatenate to "this" toString
25         return super.toString() + " Class B --> weight is: " + weight;
26     }
27 } // end of B
28
29 class C extends B{
30     private String color;
31     public C(String name, double weight, String color){
32         super(name, weight); // pass name and weight to superclass B
33         this.color = color; // color stays with "this"
34     }
35     public String toString(){
36         return super.toString() + " Class C --> color is: " + color;
37     }
38 } // end of C
```



## Method over riding

Method overriding is similar to method overloading in that we are allowed to define multiple method definitions with the same name. The details differ though.

- A overloaded method can be in the same class or a sub class and the definitions are identified by the differing signatures
  - `public void setMonth(int m)`
  - `public void setMonth(String m)`
  - The compiler can tell the difference between these two method signatures based on the call to the method.
    - `setMonth(12);`            `// calls setMonth(int)`
    - `setMonth("January");` `// calls setMonth(String)`
- An over ridden method meets the following criteria
  - Must be in a sub class
  - Must have the same signature
    - `public String toString();`    `// in super class`
    - `public String toString();`    `// in the sub class`
  - All the compiler cares about is that the syntax is correct and that there is a definition of the method available to the instance that is calling the method. It could be anywhere in the inheritance hierarchy as long as it is public.
  - The run-time environment (JVM) is then responsible for binding the call to the appropriate method definition. This is determined by the result you would get from running `getClass()` on that instance.

We have been defining over ridden methods each time we wrote our own `toString()`. The definitions we write override the definition that is inherited from the `Object` class.

You can see this by reading the Object API spec.

Method Summary		
All Methods	Instance Methods	Concrete Methods
Deprecated Methods		
Modifier and Type	Method	Description
protected Object	<code>clone()</code>	Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code>	<b>Deprecated.</b> The finalization mechanism is inherently problematic.
Class<?>	<code>getClass()</code>	Returns the runtime class of this Object.
int	<code>hashCode()</code>	Returns a hash code value for the object.
void	<code>notify()</code>	Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code>	Wakes up all threads that are waiting on this object's monitor.
String	<code>toString()</code>	Returns a string representation of the object.
void	<code>wait()</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> .
void	<code>wait(long timeoutMillis)</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.
void	<code>wait(long timeoutMillis, int nanos)</code>	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.

## The `@Override` annotation

If you are attempting to override a method you can add the `@Override` annotation to communicate your intention to the compiler. The compiler will then perform checks to ensure that your method is properly over ridden. Here is an example demonstrating this annotation with the `toString` method.

```
class C extends B{
    private String color;
    public C(String name, double weight, String color){
        super(name, weight);    // pass name and weight to superclass B
        this.color = color;    // color stays with "this"
    }
    @Override
    public String toString(){
        return super.toString() + " Class C --> color is: " + color;
    }
} // end of C
```

Take a look at what happens when we add this annotation and do not properly override a method. Notice that I added a parameter to `toString` which overloads instead of overrides. The compiler complains.

```

38     @Override
39     public String toString(int x){
40         return super.toString() + " Class C --> color is: " + color;
41     }
42 } // end of C

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
A.java:38: error: method does not override or implement a method from a supertype
    @Override
    ^
1 error

```

## Overriding the equals() Method

Here is the equals method signature from the object class

boolean	<b>equals</b> (Object obj)	Indicates whether some other object is "equal to" this one.
---------	----------------------------	---

Notice that the parameter type is Object. Up until now we have been defining our equals methods as accepting “**this**” type, which technically overloads instead of overrides. This has worked fine for our purposes but it is not inline with the rules of over ridden methods. If we were attempt to over ride the equals method in class A using A as the parameter type, the compiler would complain when we add the @Override annotation

```

1  public class A{
2      private String name;
3      public A(String name){
4          this.name = name;
5      }
6      public String getName(){
7          return name;
8      }
9      private void helper(){
10         System.out.println("Do something helpful");
11     }
12     @Override
13     public String toString(){
14         return "Class A --> Name is: " + name;
15     }
16
17     @Override
18     public boolean equals(A a){
19         return this.name == a.name;
20     }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
A.java:17: error: method does not override or implement a method from a supertype
    @Override
    ^
1 error

```

To properly override this method the parameter type needs to be Object. Remember that any Java class is a sub class of Object, which further means that instances of these classes **are Objects**, and as such can be assigned to a variable of type Object.

```

17     @Override
18     public boolean equals(Object obj){
19         return this.name == obj.name;
20     }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

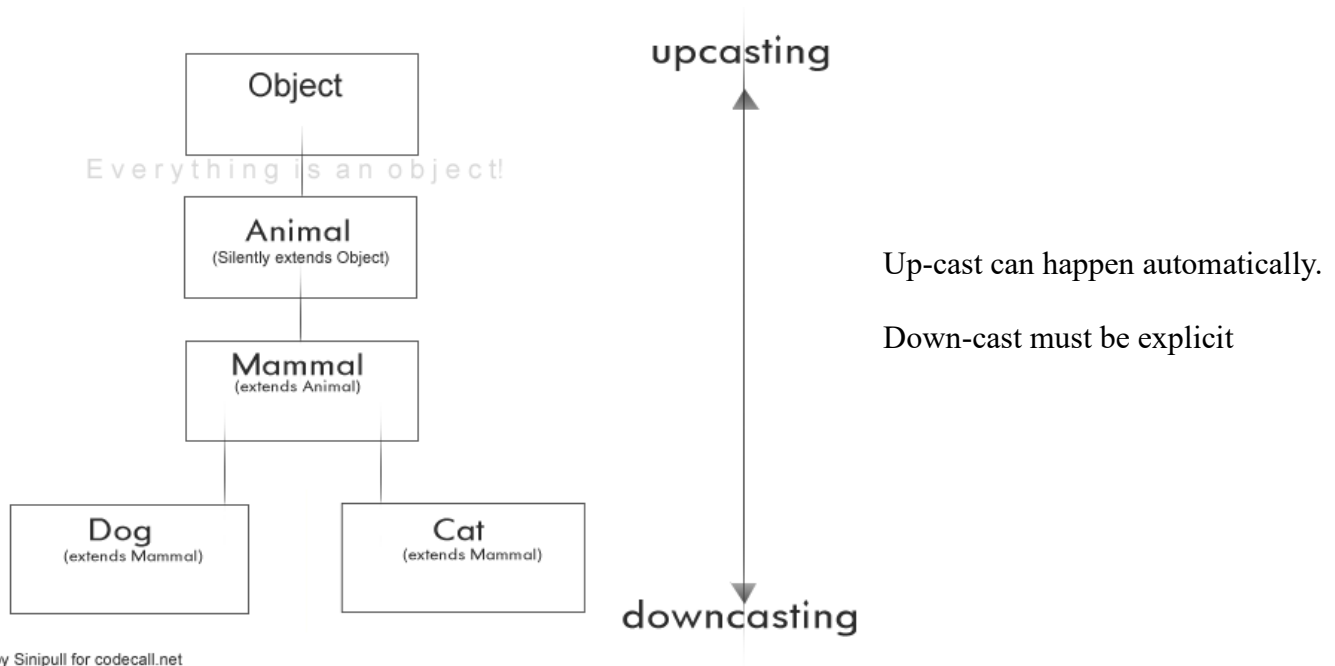
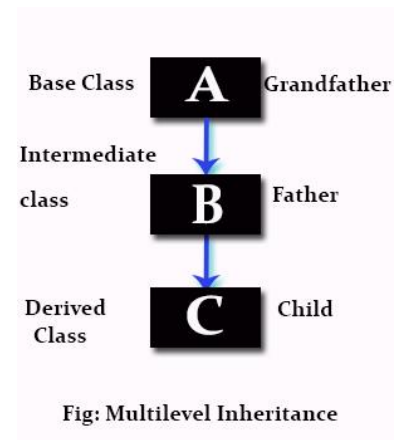
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/inheritance$ javac A.java
A.java:19: error: cannot find symbol
    return this.name == obj.name;
                        ^
symbol:   variable name
location: variable obj of type Object
1 error

```



Changing the parameter type to Object removes the error with the `@Override` annotation but now we have a different error. Attempting to reference the **name** field in Class A through a variable of type Object is not allowed as **there is no field “name”** in the Object class.

This brings up an important aspect of inheritance. A subclass object has multiple types and can be treated as an instance of any of the types above it in the hierarchy. You can pass an object of a sub-class type into a method that accepts arguments of a super class type. This is due to the “**is a**” relationship between the levels. Because C is also an instance of B you can assign instances of type C to variables of type B. You can also assign instances of type C to variables of type A . . . and by extension you can also assign instances of **any Java class** to variables of type Object, as Object is the ultimate ancestor. This is called automatic type promotion, or colloquially an **upstream cast**.



When an up-cast happens the instance of the sub-class type **becomes** an instance of the super-class type and “**forgets**” that it once was a sub-class type. In practice this means that **you can no longer call sub-class methods from the super-class variable unless the method signatures are shared (overridden)**. This is what is happening with our argument to the equals method. An instance of any Java class can be passed into a method that accepts an object, but now it can only be treated as an Object, unless we down-cast it back to its original type. Down casting is a trick operation and must be handled with precision and care.

**Rule:** You can only down cast an object if it was originally an instance of the target type.

## Instance Of vs Get Class

Before you downcast something you need to test to see if it can in fact be down casted. There are two ways to check an instances origin: **instanceof** and **getClass()**. These were described and demonstrated above. Which one should we use?

**instanceof:** Will return true if the instance in question fits anywhere within an inheritance hierarchy

**getClass():** Will tell you which class was used to build the original instance. Essentially, which type constructor was used to build the object.

```
@Override
public boolean equals(Object obj) {
    // identity check
    if (this == obj)                return true;
    // null check
    if (obj == null)                return false;
    // origin check
    if (getClass() != obj.getClass()) return false;

    A other = (A) obj;              // down cast
    if (name == null) {              // null checks
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}
```