6. Class Inheritance

Contents

[1. Introduction 1](#_Toc29835)

[2. Getting Started with Inheritance 1](#_Toc25134)

[3. References to Derived Class Instances 1](#_Toc14147)

[4. Field Hiding 2](#_Toc3962)

[5. Method Overriding 2](#_Toc11294)

[6. Improving CalcEngine with Inheritance 2](#_Toc9027)

[7. Object Class 2](#_Toc24532)

[8. Object References 2](#_Toc26773)

[9. Equality 2](#_Toc26020)

[10. Summary 3](#_Toc28788)

# 1. Introduction

=>slides: Pg. 1

Welcome to our next module, Class Inheritance.

=>slides: Pg. 2

Now as we design classes for application, those classes, of course, are going to have functionality and features. And oftentimes, we may want to design other classes that leverage those existing classes and features, and that's where class inheritance comes in. So in this module, we'll start out with an overview of inheritance. We'll see how the leverage inheritance and declare one class to inherit from another. Now when using inheritance, we have what's known as the derived class and the base class. The derived class inherits from its base class. Obviously, there's going to be a close relationship between those two classes. So the next thing we want to understand is just what is that relationship between the derived class and its base class? Now as we declare our classes, those classes have members. And in some cases, a derived class may hide the members of its base class. In other cases, it may override members in its base class. So we want to understand each of the situations and how they affect our code, Then the next thing we'll look at is Java's Object class, and the Object class plays a really important role in class inheritance. So we'll take a look at just what the role of the Object class is and how we can use it. And then we'll finish up with a look at equality checks. And the question is, if you have two instances of a class, what does it mean for those instances to be equal? Well, the answer is, it kind of depends, So we'll finish up this module by understanding the behavior of equality checks and how we can implement those equality checks.

# Getting Started with Inheritance

=>slides: Pg. 3

As we mentioned, Java allows one class to inherit from another. So the way we do that is by using the extends keyword in our class declaration. So we'll declare the name of our new class and then say it extends its base class. Now this new class, what we call the derived class, will have the characteristics of its base class. So we'll start out with the same features and functionalities that are contained within the base class. But then this derived class can add specialization. So it can build on those inherent features, then add custom features of its own.

=>slides: Pg. 4

So let's see what this looks like in code. Let's declare a new class called our CargoFlight class and we want it to inherit from our flight class so we'll say the CargoFlight class extends our flight class. Now CargoFlight can add its own members. So the first thing we'll do is have a field here, maxCargoSpace. So in this case, we're seeing our CargoFlight class has 1000 cubic meters of cargo space. Now, in addition to our maxCargoSpace, we're going to have to track how much of that space we've used. So as a CargoFlight class, we need a way to add cargo to a flight. So we'll have a public method here, add1Package, that accepts the height, width, and depth of the packages being added. So the first thing we'll do inside this method is calculate the overall size of the package by multiplying those three values together, and before we add the cargo, we want to make sure we have space for it. So we'll call another method we'll declare in our CargoFlight class called hasCargoSpace. And this will return back true or false, indicating whether there's space. If there is space, we'll go ahead and add the size of that package to the amount of cargo space we've already used, and if there is not space, we'll call another method in our CargoFlight class called handleNoSpace.

=>slides: Pg. 5

Now let's take a look at those two methods. So we have our hasCargo Space method; it accepts the size of the package that we want to add. So all we do inside of here is take the size of that package, add it to the amount of space we've already used, as long as the result of adding those two together is less than or equal to our maxCargoSpace, it will return true. And then in the case of our handleNoSpace method, we can print out the message saying there's not enough space.

=>slides: Pg. 6

So let's see what it's like to use our CargoFlight class. Now just as a reminder, our CargoFlight class extends our Flight class. We have just one public method here, add one package. So we're going to use our CargoFlight class just like we use other classes. We'll create a new instance of our CargoFlight class, and we'll assign that to a reference of type CargiFlight named cf. Now once we have that reference, we'd of course call our add1Package message, which allows us to add a package to this instance of our CargoFlight class. So in addition to having the ability to add a package to this flight, we can also do things that the Flight class allows. So we declare an instance of a passenger, we can then use our cf reference to call the add1Passenger method that was declared in the Flight class. So, as you can see, our CargoFlight class has all the features and capabilities that we've declared as part of the CargoFlight class, but it also has the features and capabilities that were declared in the Flight class. So now in our next section, let's see other type of reference that we use affects the available features we can access.

# References to Derived Class Instances

=>slides: Pg. 7

Now when we create an instance of a class, we're generally going to want to store a reference to that class instance. And when we're dealing with classes that derive from another class, we have some interesting options because we can actually assign an instance of that class to a reference whose type is that of the base class. Now this is a really powerful concept, but it does have some implications in terms of what features are available because the features we can access are dictated not by the class instance type, but instead are dictated by the reference type. We'll only be able to access members that are visible to the type of the declared reference.

=>slides: Pg. 8

So let's go ahead and create an instance of our CargoFlight class, but let's assign it to a reference whose type is flight. So the type of our variable here, f, is the Flight class, even though it's pointing to an instance of CargoFlight, and that's valid because CargoFlight inherits from Flight. Now we, of course, can use any flight‑based features here. So I want to go ahead and create an instance of a passenger. I can use my f reference to call add1Passenger to add that passenger to the flight. And, of course, this works because the method add1Passenger is declared as part of our Flight class. But what if we try to call f.add1Package? We're dealing with an instance of our CargoFlight class, and the CargoFlight class has a method add1Package, but the Flight class doesn't. So the attempt to call add1Package with a reference whose type is Flight would actually generate an error because the methods we can access are tied to the type of the reference, not the type of the class instance. And although this may sound like a bit of a limitation, it's actually a really powerful concept because it makes it easy for us to group work that's tied to the base class without regard for any specific derive classes. So let's say we need to process a number of flights.

=>slides: Pg. 9

So we declare an array here, squadron of type Flight with 5 slots in. And for our 0 element, we're going to assign that an instance of the Flight class. But then for our next element, we'll assign it an instance of the CargoFlight class, and we can keep doing this. Now we could actually take this array and pass it the code that doesn't know anything about the CargoFlight class. You could loop through the array, do any flight base work that was necessary without regard for the specific kind of flight it is. As long as it is the Flight class or something that derives from Flight class, the code will be able to process this array. Now for those elements that are actually cargo flights, we have the option to do additional work, but we could easily group all the flight base work together here in this single array. Alright, so now in our next section, let's take a look at an issue known as member hiding.

# Field Hiding

=>slides: Pg. 10

Now as we declare our classes, those classes, of course, have members. And when we're dealing with inheritance, we have the members in the base class as well as any members we add in the derived class. And in general, the members of the base and the derived class blend together in a way that's pretty natural, but there are so special situations that we want to be aware of. One of those relates to fields because there's a scenario where a field in a derived class can hide a field in the base class. If your derived class declares a field that has the same name as a field in the base class, the derived class doesn't replace the base class field, the base class field is still there, but the derived class field is hiding the field from the base class.

=>slides: Pg. 11

Now just a quick reminder, we have our Flight class. Now as you recall, our flight class has a field named seats that we initialize with a value of 150. Our Flight class also has a method add1Passenger. This add1Passenger method wants to increment the number of passengers, as long as this method hasSeating returns a value of true. Remember, hasSeating method is pretty straightforward, it just checks to see if the number of passengers is less than the value in our seats field.

=>slides: Pg. 12

So let's say when we declare our cargoFlight class, we also declare a field named seats with the value of 12 and we know our cargoFlight class extends the Flight class and the Flight class already has a seats field whose value is 150 and this method hasSeating depends on the value in that seats field. So our cargoFlight class' seats field is hiding the seats field in our Flight class and that seats field is an important part of our hasSeating method.

=>slides: Pg. 13

So let's see what happens when we start using these classes in code. So let's create an instance of our Flight class, we'll assign it to a reference F1 whose type is flight. So if we now print out the value of F1.seats, we get exactly what we expect, the value 150 because that's the value we set seats to in our Flight class. Then let's create an instance of our cargoFlight class, we'll assign that to a cargoFlight reference named CM. So if we print out cf.seats, again, we get what we expect, the value 12, the value we set seats to in our cargoFlight class. So at this point, things look pretty natural. It seems that the seats field in our cargoFlight class has replaced the seats field in our Flight class, but that's actually not the case. The seats field in cargoFlight is simply hiding the seats field in our Flight class, and we can verify that that's the case. Let's go ahead and create another instance of cargoFlight, but this time we're going to assign it to a reference named F2 whose type is flight. So we have an instance of our cargoFlight class being accessed by a flight reference. So when we print out F2.seats, we don't get the value 12, we get the value 150 because this code, which is using a flight reference, is using the values of the seats field as it's defined in the Flight class, even though the type of our instance cargoFlight has declared a field with that same name, and this can create some really ugly problems. For example, if we call f2.add1Passenger, remember, the type of F2 is flight, but it points to an instance of cargoFlight. So when we call add1Passenger, which is implemented in our Flight class, it uses the method hasSeating, which is also implemented in the Flight class. So when hasSeating accesses the field, seats, it's accessing the version of seats that's declared in our Flight class, so it's going to use the value 150. So even though our cargoFlight has set a field named seats to a value of 12, our hasSeating method is using a version of the seats field that was set to 150, and this problem is not tied to the type of reference we use because even if we use cf.add1Passenger, remember that CF is a reference of type cargoFlight that points the cargo flight. When it calls add1Passenger, which is implemented in our Flight class, which then calls hasSeating, which is implemented in the Flight class, it's still going to use the version of the seats field that's defined within the Flight class, which means it's going to use that value of 150, so obviously, this is a very concerning situation. So in our next section, let's see a better solution to dealing with this kind of issue.

# Method Overriding

=>slides: Pg. 14

Now, as we've seen, if a derived class declares a field with the same name as a field in the base class, the field in the derived class hides the field in the base class, but it turns out when it comes to methods, things behave very differently. In the case of methods, if the derived class declares a method with the same signature as a method in the base class, the derived class overrides the base class method. In other words, the method in the derived class replaces the method in the base class. So let's see how we can use this behavior to improve our Flight and CargoFlight classes.

=>slides: Pg. 15

So as you recall, our Flight class has a field here named seats with a value of 150.

=>slides: Pg. 16

Well, let's change this from being a field to a method named getSeats, which returns 150.

=>slides: Pg. 17

Then down in our hasSeating method, we'll change that to use our getSeats method rather than our seats field,

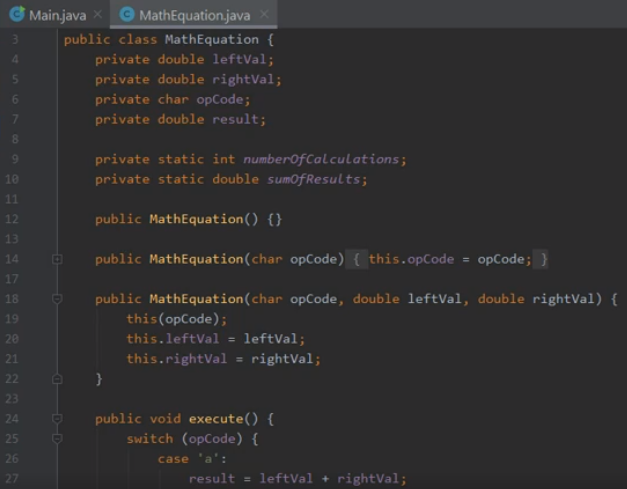
=>slides: Pg. 18

and then over in our CargoFlight class, we'll override the getSeats method and instead have that return the value of 12. So now our Flight class declares a method named getSeats, and our CargoFlight class overrides that method.

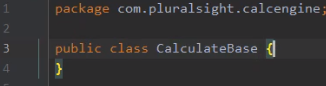
=>slides: Pg. 19

So now let's see how this change impacts the behavior of our classes. So we'll start out by creating an instance of our Flight class, and we'll assign it to a reference named f1 whose type is Flight. So if we now print out f1.getSeats, we get the value we expect, which is 150. If we then create an instance of our CargoFlight class, assigning that to a reference named cf whose type is CargoFlight, if we print out cf.getSeats, again, we get what we expect, which is the value of 12. Now so far, our behavior is very much like that of when we were using fields. But now let's take a look at this example. Let's create an instance of CargoFlight and assign it to a reference f2 whose type is Flight. And then we'll print out f2.getSeats. So now in this case, we're using a reference whose type is Flight, pointing to an instance whose type is CargoFlight, and in this case, now we get the value 12. Notice that we're using the implementation of getSeats that's tied to the class instance type, not the reference type, and that's a key difference between methods and fields. In the case of fields, it's the type of the reference that determines which version of the field you use, but when it comes to methods, it's the type of the instance that determines which method you use. So here, in this case, even though the type of our reference is Flight, because the instance is a CargoFlight, we use the implementation of getSeats provided by the CargoFlight. And this is a really powerful concept. We can see that when we do something like cf.add1Passenger. Remember that cf's type is CargoFlight, pointing to an instance of CargoFlight. We call add1Passenger, which is implemented in the base class Flight. It then calls hasSeating, which is also implemented in the Flight class. And remember that hasSeating calls the getSeats method. Well, because this is an instance of CargoFlight, when hasSeating calls getSeats, it calls the implementation of getSeats within CargoFlight, which means the hasSeating method will use the value 12. So even though the hasSeating method is implemented in the Flight class, it knew to use to the getSeats implementation from the CargoFlight class. And this is true no matter what kind of reference we use. So if we say f2.add1Passenger, again, remember, the f2's is Flight, but it points to an instance of CargoFlight. So when we call add1Passenger, it calls hasSeating, and again, because this is an instance of CargoFlight, when hasSeating calls getSeats, it calls getSeats from CargoFlight, which again means that hasSeating uses that value of 12 when doing the comparison. So to help us get a better understanding of all this, in our next section, let's get back into STS, and we'll see how we can use inheritance to improve our calc engine project.

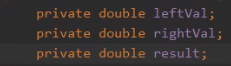
# Improving CalcEngine with Inheritance



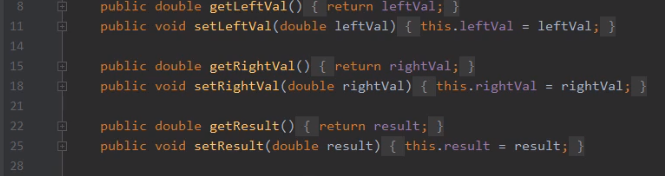
Here we are in STS, again, looking at our CalcEngine project. And what we want to do now is see how we can use inheritance to simplify our application and actually make maintaining the code a bit easier. And as you recall, the bulk of the work in our CalcEngine project is done with this MathEquation class. It gives us the ability to do the four basic math operations. So now to do that, the MathEquation class has some fields here. So we have our leftVal and rightVal fields, which are both doubles. We have a result field, which is a double, then we also have this opCode, which is a char. Remember, the opCode indicates what operation we want to perform. So using that opCode, the bulk of the work is actually done in the MathEquation class' execute method. So let's scroll down and take a look at that. So here we are now at our execute method. And as you recall, the bulk of the work in the execute method is this switch statement. We need to have this switch statement because this execute method doesn't really do just one thing. It actually does four different things. And so based on the opCode, the execute method has to choose which of those things to do and then perform the appropriate operation. And although there's nothing wrong with this approach, but as we'll see by using inheritance, we can actually allow the JavaType system to make some decisions for us, which will allow us to actually write code that's much simpler to maintain. So to do that, let's start out by creating a new class that'll service the base class for our calculations.



So to create that class, I'll go over here to the project window. I'll, again, right‑click on our package name. I'll choose New, and then we'll go and say Java Class. For the name of this class, let's call it CalculateBase. So to create our CalculateBase class, I'll go ahead and hit Enter. So that gives us our CalculateBase class. So what we'll want to do here now is add in some of the fields that we'll need.



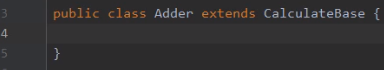
So let's add fields for our leftVal, rightVal, and result. So now we have those three fields in place, and we've marked them all as private. And, remember, by marking them as private, it means those fields are not accessible outside this class.



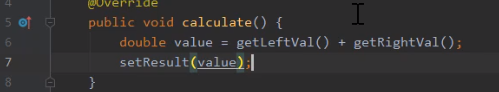
So that means that in addition to the field, we'll also need to have getters and setters for each of those fields. So now we have our getters and setters in place, so let's go and add a method we'll use to do the calculations.



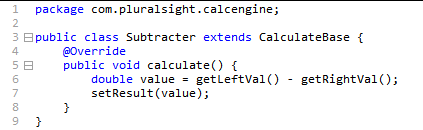
So let's create a public void method named calculate. So that gives us our calculate method, but we're not going to do any actual calculations in this class. We're going to actually have classes that derive from this that will handle those details.

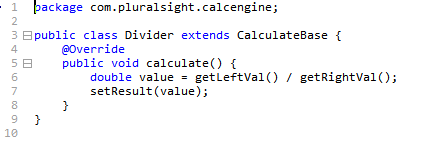


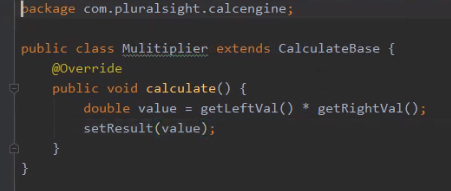
So let's add another class to our project named Adder. So, again, over here in our project window, we'll go up to the package name, and we'll right‑click, choose New, and then we'll choose Java Class. So this class will be used to do our additions, so I'll name this class Adder. Then I'll go ahead and hit Enter. Now we want our Adder class to extend CalculateBase. So that means that Adder will have all the characteristics of CalculateBase. So all we need to do is incorporate the specializations that we need for this Adder class. What we want to do is specialize the calculate method.



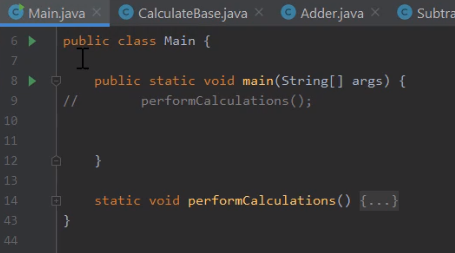
So let's, again, declare our calculate method. Now, remember, we want this calculate method to override the calculate method in CalculateBase. So let's go ahead and mark it with the @Override annotation. So now the work we want to do in this calculate method is really, really simple. We simply want to add our left value and right values together. So let's start out by declaring a local variable named value of type double, and we want to add the left values and right values together. But remember, the leftVal and rightVal fields are private in CalculateBase, so we can't use the field directly, but we can use the getters. So now we get our left and right values, add them together, and then store the sum and value or local variable. So now what we want to do is set the result field, so we'll call setResult, passing in value, and that's all there is to it. The Adder class is now a specialization of CalculateBase that knows how to handle the details of adding two values together.

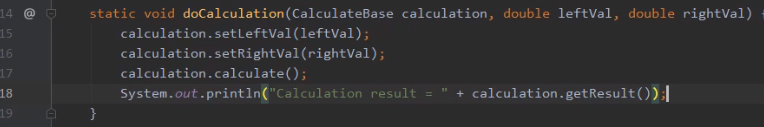




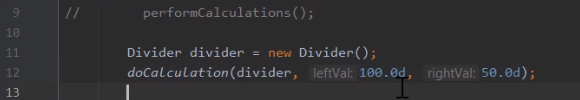


So I'm going to add three more classes to our project. I'll add Subtractor, Divider, and Multiplier. So now we have these four classes that all inherit from CalculateBase and do specific math operations. So let's see how we can use these classes. So let's head over here to our Main class.

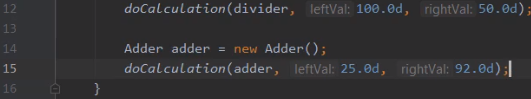


And just after our Main class' main method, let's add a method named doCalculation.  


Now the first parameter to doCalculation will be a reference to CalculateBase named calculation. And then we'll also pass in a leftVal and a rightVal. And then here inside of doCalculation, we want to do the work it takes to perform a calculation. So the first thing we'll do is use our calculation reference and set its leftVal and rightVal. And then once we set those two values, we're going to want to perform the calculation, so we'll call the calculate method. And then once we calculate, we'll go ahead and print out the result. So now as we look at this code, it has all the steps we need to perform a calculation, but there's one thing that's kind of curious. When we call calculate on CalculateBase, we know it doesn't actually do anything. The implementation of calculate on CalculateBase was just an empty method, but that's okay because, remember, there are classes that inherit from CalculateBase to actually override that method, and we can actually use instances of those classes. So let's head up here to our main method, and we'll add some code that does that.



So let's start out by declaring an instance of our Divider class. And then once we have our instance of divider, let's call doCalculation, passing in divider and then two values, 100 and 50. So what happens now is we know that our Divider class overrides the calculate method to perform division. So when we call doCalculation, passing in divider, even though doCalculation receives that into a CalculateBase reference, when that reference calls calculate, it will use the divider implementation. Therefore, we should print out the result of dividing those two values together. So let's do one more of these kinds of calculations.



Let's go ahead and create an instance of the Adder class. Then we'll call doCalculation, passing in Adder along with the two values, 25 and 92. So now we have our code in place that will do calculations using a divider and using our adder. Let's go ahead and run our code, and let's see what happens.



If we look down here at our Run window, we can see our results. Well, the first calculation we performed was dividing 100 by 50, and we can see our result is 2. So our Divider class worked. Then our second calculation was adding 25 to 92, and we could see the result is 117. So our code is using inheritance to achieve the exact result we were looking for. Alright, so now in our next section, let's take a look at the Java Object class.

# Object Class

=>slides: Pg. 20  
Let's take a look now at the Java Object class. The Java Object class is really important when we talk about inheritance because the Object class is the root of the Java class hierarchy. Every class we declare has the characteristics of object, and this is really important when it comes to talking about references because an object reference can reference any class instance. In fact, an object reference can even reference an array instance because in Java, arrays are a type of class.

=>slides: Pg. 21  
And the thing to keep in mind, when we're talking about inheritance, every class in Java inherits either directly or indirectly from the Object class. So let's take a look at some of the classes that we've declared. Now, of course, we've talked a lot about our Flight class. Well, if we were diagramming the classes we've created for our Flight class, we might just draw a box like this. And then when we get to our CargoFlight class, our CargoFlight class extends the Flight class, meaning that our CargoFlight class inherits from the Flight class. So to diagram this, we would draw a box for our CargoFlight and then draw an arrow up to the Flight class, and that indicates that CargoFlight inherits from our Flight class. Now looking, again, at our Flight class, our Flight class doesn't appear to extend any other class. But in Java, if a class doesn't explicitly extend another class, then it implicitly extends the Object class. So as we're diagramming our class hierarchy here, we should indicate that our Flight class is inheriting from Object. Now as part of declaring our Flight class, we could explicitly write extends Object. But there's really no reason to do that because by not indicating that we extend anything, we automatically extend the Object class, and this is true for any class we declare. So when we declare our Passenger class, we don't indicate that it extends any class, so it automatically extends the Object class. So this shows us that every class inherits from the Object class, and this has important implications when it comes to dealing with object references. We'll take a look at that in our next section.

# Object References

=>slides: Pg. 22  
Now as we mentioned, every class inherits either directly or indirectly from the Object class, and that has important implications when it comes to object references, because it means a reference of type Object can point to any class instance. And this is useful in scenarios where you might need to hold on to some otherwise unrelated types. So for example, if we declare an Object array here named stuff and we allocate out three slots in the array, well, stuff[0] might point to an instance of the Flight class, stuff[1] might point to an instance of our Passenger class, and stuff[2] might point to an instance of our CargoFlight class. Now when working with our stuff array, we couldn't deal with any of the type‑specific capabilities of any of these classes, but we are able to hold on to the references, and even pass them around to methods or other parts of our code. And object references give us a great deal of versatility, so let's go ahead and declare another Object reference here named o, and we'll point it to an instance of our Passenger class. Now because the type of o is Object, we can actually assign an entirely different type to it. So we can say o = and create a brand new instance of a Flight array. And notice what we're doing here. We've created an array named Flight, but we're assigning it to a variable o that itself is not an array. It's simply an object reference. And a key thing to understand here is that in Java, the array itself is considered a type of class. So the Flight array is a type of class, and that array holds references to another type of class named Flight. So looking at these two pieces of code, on the left, we have an array whose type is Object, which each element in the array pointing to an instance of a particular class. On the right, we have this variable o, whose type is Object, that holds a reference to a class, which happens to be an array.

=>slides: Pg. 23  
But now something to keep in mind, although an object reference can point to an instance of any class type, that object reference can't access the capabilities of that class type. So for example, say we created a variable here named o, whose type is Object, and we assign to it a reference to an instance of our CargoFlight class. So that means we have that CargoFlight instance with a reference to that instance stored in o. Now we know that the CargoFlight class has a method named add1Package. Well, if we try to say o.add1Package, even though CargoFlight exposes that method, remember we can only access features that the reference itself knows about. And since the object reference doesn't know anything about the features in the CargoFlight class, we would actually get a compilation error when we try to call add1Package. Now this doesn't mean that CargoFlight doesn't have all these capabilities. It simply means that the reference named o doesn't know how to access the CargoFlight capabilities. So what we could do is actually assign that reference o over to a reference named cf of type CargoFlight. Now this line, as it's written here, wouldn't actually compile. Remember that the type of o is Object, and it can point to any class type. So the compiler can't be sure that o will always point to an instance of CargoFlight. So to deal with that, we need to include a cast in this assignment. And basically here, what we're telling the compiler is that trust me, I'm sure that o points to an instance of CargoFlight, so go ahead and assign that reference over to cf. So by doing that, we would allocate out the reference named cf, then we would take the reference that's held in o, pass that over into cf, and now cf holds a reference to that same instance, which means that now we can call cf.add1Package, and that code would work just fine. But something to be very careful about, the line where we assign o to cf, if o happens to point to something that's not a CargoFlight, we would actually crash at runtime. So in order to program effectively, we should verify that o actually holds a reference to CargoFlight. And we can do that with an if statement that uses the instanceof operator. What the instanceof operator does is it takes a look at the reference that o holds, and it makes sure that the object it points to is an instanceof CargoFlight. If that's not the case, instanceof will return false, and we would skip over the code that actually assigns o to cf.

=>slides: Pg. 24  
Now as we've mentioned, a reference of type object can't access any of the specific behaviors of our class types. But there are some behaviors that are common to all classes because the Object class does expose certain methods. For example, one of the methods we have is what's called the clone method, and this allows us to create a new instance of an object that's a duplicate of the current instance. There's a method called hashCode that returns back a hash code for a current instance. And this is useful for scenarios that require a hash code, things like hash maps that we use to store collections of objects. There's a method getClass that will return type information for the current instance. There's a method called finalize that we don't really override very often, but this handles special resource cleanup scenarios that occur in some very specific scenarios. There's a toString method that will return back a string representation of the current instance. And then finally, we have the equals method. And the equals method allows us to compare two instances to see if they're actually equal. And we'll take a closer look at equals in our next section.

# Equality

=>slides: Pg. 25

Let's take a look now at the issue of equality. If I have two object references, what does it mean for those references to be equal? And the answer is, well, it kind of depends. It depends on specifically what you're looking for in terms of being equal. So to demonstrate, let's start out by creating a new instance of our Flight class and setting a flightNumber of 175. We'll reference that instance in a variable f1. Then we'll create another instance that also has a flight number of 175, and we'll assign that to a reference named f2. So now the first question is, how do I compare them for equality? Well, one option would be to use the == operator. When we compare primitive types such as integers, that's the operator we use for equality. So what happens if we use this with references? In Java, if you use the == operator with references, what it actually does is compare the references themselves. So it looks in each reference and says, do both of these references point to the exact same object instance? And, of course, in our case they don't. They point to two separate objects. So this comparison will result in false. So now what's our next option? Well, remember that every class inherits from the Object class. And we said the Object class has an equals method, so maybe we can use that method. So I'll do a logical test using f1.equals, passing in f2. But, of course, the problem here is that the full implementation of the equals method that we inherit from the Object class does the same comparison as the == operator. So, again, it checks to see if both references point to the exact same object instance. So, again, we would get back false. And the issue here is that we're always looking at the references. In general, when working with class types, when we do equality comparisons, we actually look at the contents of the objects. We want to look at the values within each object instance to determine if those instances are equal. And the good news is we can do that.

=>slides: Pg. 26

So looking at our Flight class, we know we have a number of fields, but two of those fields were linked to identifying a flight. If you remember in one of our earlier modules, we said we could identify a flight in one of two ways, either by using the flight's flightNumber or the flight's Flight class. We set one or the other, but not both. So if you create a flight with a flightNumber of 175, the flightNumber field will have that value 175, Flight class will still have its default value. So one definition of equals we might use for the Flight class is to compare these identifying fields. So to do that, we'll override our equals method because, remember, we can override any method we inherit. Now one quick note on overriding methods. Remember that when we do an override, we have to match the signature exactly of the method we're overriding. And believe it or not, sometimes that's harder to get right than you might think. Any small change in the way the method is declared might not actually override that method, but instead be declaring a brand‑new method. So to help us with that, Java provides the @Override annotation. So by putting this @Override before the method, that tells Java it's my intent to override a method that I've inherited. So a compiler will verify that for us. So now here in the body of our equals method, one of the first things we'll need to do is take that object reference that we received and cast it to be a flight reference because we want to look at the flight aspects of the object we're comparing to. And then once we have that reference, we can do the comparison. Now something that's important to understand, there is a one‑universal definition of how you compare every type of class for equality. It's up to you as a class implementer to decide the right way to determine what equality means. We're going to make the determination that we consider two flights to be equal if they have the same identifying information. So what we'll do here in our equals method is we'll simply return back the result of comparing the current instances' flight number to the passed in instances' flight number and comparing the Flight class of the current instance to the Flight class of the passed‑in instance. So basically, in our implementation of the Flight class, two flights are equal if they have the same identifying information.

=>slides: Pg. 27

So now with that work done, let's see how our equality comparisons work. Now, of course, we can still do the f1 == f2, but that == operator is still going to compare the references themselves. So that comparison is still going to return false, but now when we call f1.equals, passing in f2, it uses our override of the equals method. So it's actually going to compare the contents of those two instances, and that will give us back a result of true. So now our Flight class can handle equality comparisons of two flight instances. But there's another situation we need to think about. What if someone creates an instance of the Passenger class and then calls f1.equals, passing in a reference to that Passenger class? What's going to happen? Well, it turns out, at that point our code would actually crash. So we need to deal with this scenario as well.

=>slides: Pg. 28

So back here in our Flight class, when someone passes in a reference whose type is Passenger, we're going to receive it here in our o parameter whose type is Object. Then inside the equals method, the first thing we're going to try and do with it is cast that o to be a flight, but o doesn't point to a flight. It points to a passenger, so that's not a valid cast, and that's why our program would crash. So we need to protect against this. So to do that, we're going to use that instanceof operator. And remember that instanceof makes sure that a reference points to an object of an appropriate type. So as long as o points to an instance of the Flight class, everything is fine. So what we're going to do is apply a not to that comparison. So this sort of statement will be true if o points to anything that's not an instanceof Flight. And if that's the case, we're simply going to return false, which will handle the scenario of receiving a reference to something that's not a flight.

=>slides: Pg. 29

So if we, again, go through the equality checks, the == operator will still be false, f1.equals passing in f2 will still be true, but now when we create an instance of passenger and then call f1.equals, passing in the reference to the passenger, now we get back false. So now we have an equals implementation that properly compares two flight instances, as well as handles any references whose type is not a flight.

# Java Beans

=>slides: Pg. 30

JavaBeans are [classes](https://www.geeksforgeeks.org/classes-objects-java/)that [encapsulate](https://www.geeksforgeeks.org/encapsulation-in-java/) many objects into a single object (the bean). It is a java class that should follow following conventions:

1. Must implement [Serializable](https://www.geeksforgeeks.org/serialization-in-java/).
2. It should have a public no-arg constructor.
3. All properties in java bean must be private with public getters and setter methods.

**Syntax for setter methods:**

1. It should be public in nature.
2. The return-type should be void.
3. The setter method should be prefixed with set.
4. It should take some argument i.e. it should not be no-arg method.

**Syntax for getter methods:**

1. It should be public in nature.
2. The return-type should not be void i.e. according to our requirement we have to give return-type.
3. The getter method should be prefixed with get.
4. It should not take any argument.

For Boolean properties getter method name can be prefixed with either “get” or “is”. But recommended to use “is”.

# Java Beans Example

=>slides: Pg. 32

The requirements for an object to be a bean is to define a public parameterless constructor, so that beans can be instantiated by builder tools in an uncomplicated way . Secondly, one of the interfaces java.io.Serializable or java.io.Externalizable need to be implemented. The interfaces do not prescribe the implementation of any methods, but are an approval, that the bean may be saved in persistent storage as in a file or database. In doing so, the bean can be restored after an application was shut down or transferred across networks. The ability to store state of a component in persistent storage is called *persistence*. Java offers a standard serialization mechanism, which makes it very easy to serialize objects.

# Summary

=>slides: Pg. 30

To wrap up, here's some of the key things you want to remember from this module. Remember that as we declare our classes, one class can inherit from another. And this provides some really powerful capabilities because remember that the derived class will have the characteristics of the base class that it inherits from, but then the derived class can actually add its own specialization, so the derived class builds on the work that was done in the base class.

=>slides: Pg. 31

Now remember that inheritance affects the type of references we can use. When one class inherits from another, we can actually assign an instance of the derived class to a reference whose type is that of the base class. So base class references can point to instances of the derived class. But something to keep in mind, when using that base class reference, the only features we will have access to are the features that the base class is aware of. Remember the type of the reference affects which features can be accessed. But then the derived class can override methods that it's inherited. Now remember, to do that, the derived class's method must have the same signature as the base class method, and Java gives some help to ensure that. Remember we have that @Override annotation. We apply that to a method to tell the compiler it is our intent to override a method that we've inherited. And then by overriding that method, remember that that method now becomes implementation used for all instances of your class, even if your class instance is referenced by a reference whose type is that of the base class. So even when using a base class reference, the overridden version of the method is what gets used.

=>slides: Pg. 32

Then we looked at the Object class. Remember the Object class is the root of the Java class hierarchy. So that means that every class in Java inherits either directly or indirectly from the Object class, which means that every class in Java has all the characteristics of the Object class. And the Object class provides a number of methods that we commonly override, which allows us to specialize those methods for our particular class. One of those that we most commonly override relates to checking for equality because remember there's a couple different ways to check for equality. One way is to use the equality operator, or the == operator, but remember that does a reference‑based comparison. In other words, it checks to see if both references point to the exact same object instance. In general, that's not what we want. In general, we want to compare the contents of each object to see if they're equal. So to do that, we're going to override the equals method that we inherited from the Object class. And within that equals method is where we'll compare the contents of our classes to determine what we consider to be equal. Alright, that wraps up this module. In our next module, we'll continue our discussion of inheritance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*