

Unit II: Creating Code

Chapter 14: Class Procedures: Reusable Code

In this lesson, you will create class procedures. Class procedures let you write code for a class of objects to perform an action. For instance, you can write a procedure for a biped to wave. Then, whenever you want a biped character to wave, you just call the procedure, just like you would use the say and think procedures. The code can then be used many times and for many objects of the same class. This will allow you to decompose a program into functional subprograms and create and test larger programs. We will also use parameters so that we can tell the procedure how fast to wave, or how many times to wave. We can also use parameters to tell who the bunny faces when it waves. This is a very powerful tool and will let you greatly improve the efficiency of your code.

Objectives

- Write code to animate a character, then convert that code to a procedure so that it can be used many times and for other characters.
- Utilize **parameters** for flexibility.
- Use parameters so that the same procedure can vary in speed, repetitions, or distance.
- Use parameters to pass an object, such as a marker, or another object that you want to use in the procedure.
- Decompose a program into subprograms: **myFirstMethod** will be mostly calls to procedures and functions from now on.

Waving

In the previous lesson, we wrote code to make Yeti wave. The code was specific to Yeti. Now we have 4 characters, and we want to make them all wave, say hello, do a little dance and wave goodbye. Repeating that waving code for each of 4 characters twice (once to say hello and once to say goodbye) would mean that we repeat pretty much the same code 8 times. That would be tedious. Fortunately, there is a better way. We will copy all the code to the clipboard, create a biped procedure and drag the code from the clipboard, then just change the name from yeti. Open the Yeti Waving program and save it using a new name.

Yeti Waving

```

public void myFirstMethod() {
    // Yeti Waves
    // Programmer : Janet Joy
    Double turnBack = 0.2;
    // Get arm into position
    doTogether( () -> {
        this.yeti.say( "hello", Say.duration( 10.0 ) );
    }, () -> {
        this.yeti.getRightShoulder().turn( TurnDirection.LEFT, turnBack );
    }, () -> {
        this.yeti.getRightElbow().turn( TurnDirection.BACKWARD, turnBack );
    } );
    // Wave three times
    Double waving = 0.16;
    Double turnRight = 0.16;
    this.yeti.getRightElbow().turn( TurnDirection.LEFT, waving );
    for( Integer indexA = 0; indexA < 3; indexA++ ) {
        this.yeti.getRightElbow().turn( TurnDirection.RIGHT, turnRight );
        this.yeti.getRightElbow().turn( TurnDirection.LEFT, waving );
    }
    // Back to straight arm
    doTogether( () -> {
        this.yeti.say( "Bye" );
    }, () -> {
        /* disabled
        this.yeti.straightenOutJoints();
        */
    }, () -> {
        this.yeti.getRightShoulder().turn( TurnDirection.RIGHT, turnBack );
    }, () -> {
        this.yeti.getRightElbow().turn( TurnDirection.FORWARD, turnBack );
    } );
}

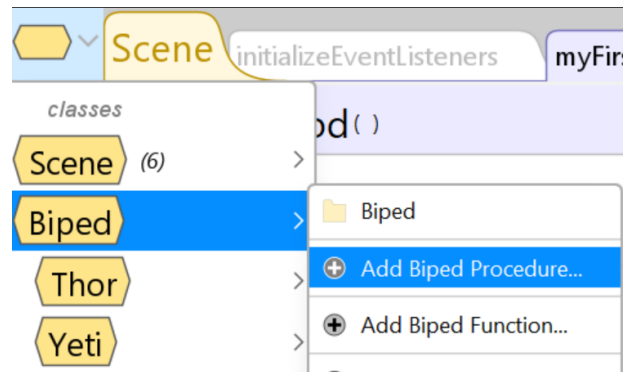
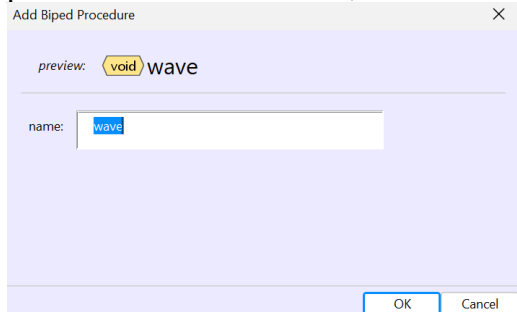
```

Clipboard

To make it easier to copy the code, we will put a **doInOrder** block in the program and drag everything that is part of waving into the **doInOrder** block. This doesn't change the sequence, but check that the program still works, just in case. Right click on the whole **doInOrder** block and copy to clipboard.

Biped Procedure

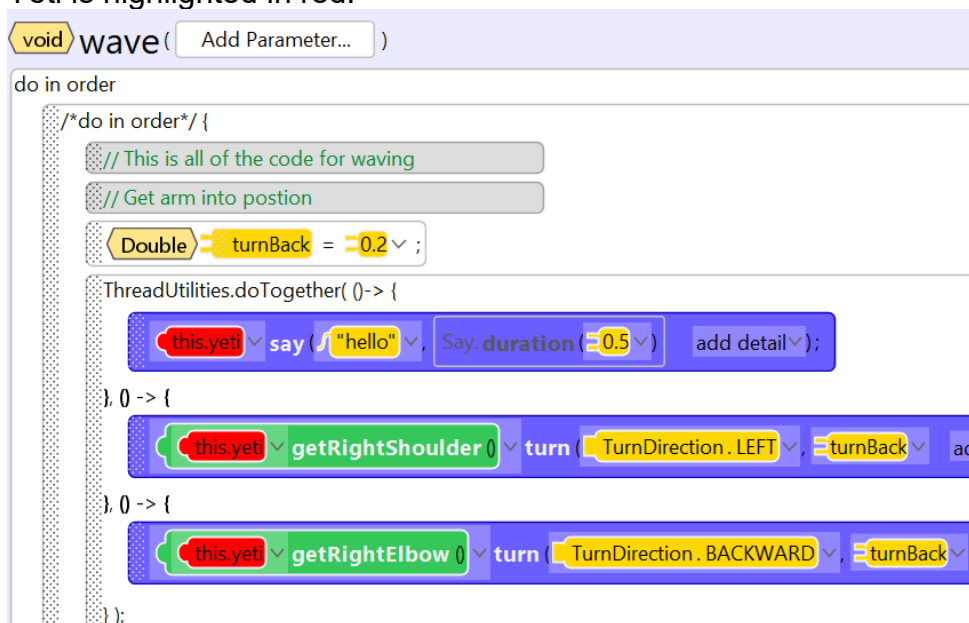
With Yeti selected, Click the 6-sided yellow shape at the top of the code view. Select Biped, then select Add Biped Procedure. A dialog window opens to name the procedure. Name it **wave**, then click OK.



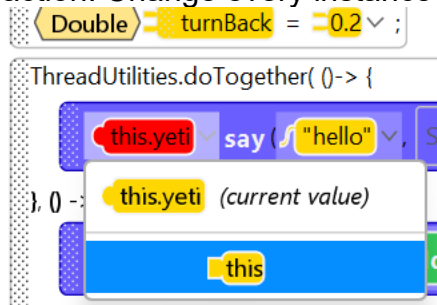
A code window opens: Drag the clipboard into the code window.



After you drag the clipboard into the wave procedure, you will see everywhere you used Yeti is highlighted in red.



The biped procedure **wave** is going to work for every biped, not just Yeti. In a biped procedure, the key word **this** will be used to specify the biped that will perform the action. Change every instance of **yeti** to **this**.

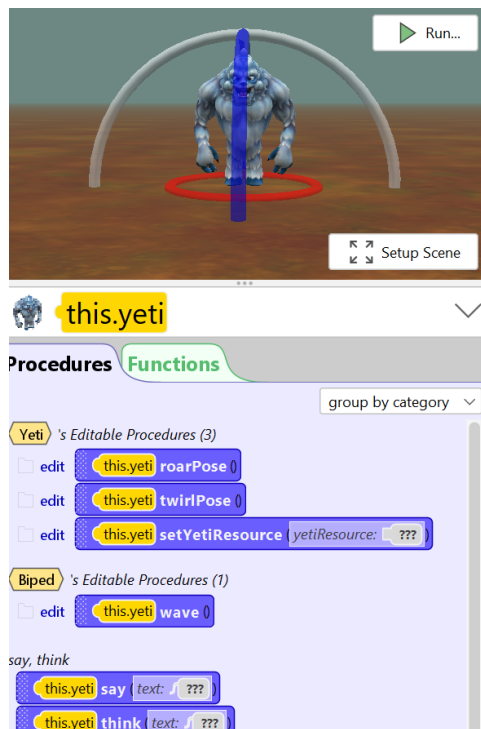


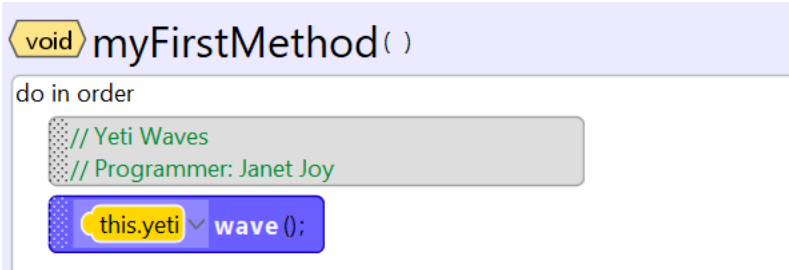
We can also drag all the code out of the **doInOrder** block, we only need that to make it easier copy the code to the clipboard.

myFirstMethod

Go back to **myFirstMethod** and delete all of the code for waving:
(No worries: that's why we saved the program to a new name!)

Select Yeti. Now, just before the say and think tiles there is a new tile, **this.yeti wave**. Drag that yeti wave into the code window.





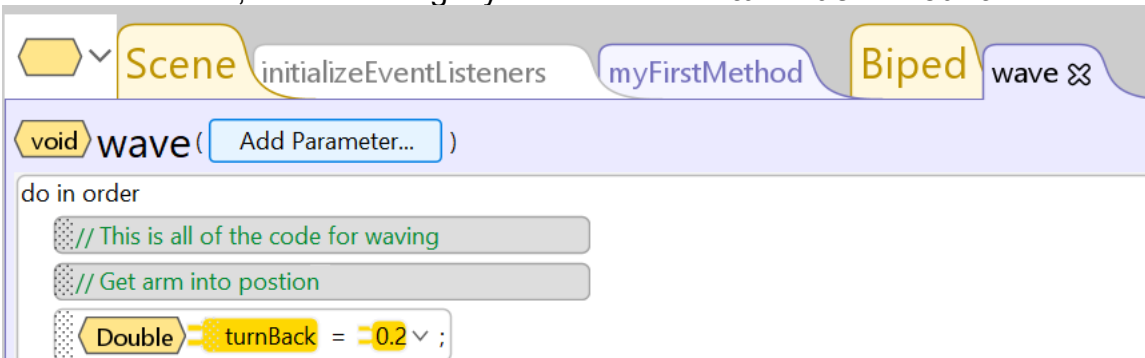
Add a few other characters and make them all wave. Make them all wave together. Make them all wave 3 times!

Parameters

If you make them all wave together, you will notice that they all wave in unison. In real life, some people would bend their arms more, wave faster, or more times, etc. We can add parameters to the wave procedure, then use different arguments so that there is variety in the wave. We already have some variables for the amount to bend the arm and shoulder; we just need to use parameters instead.

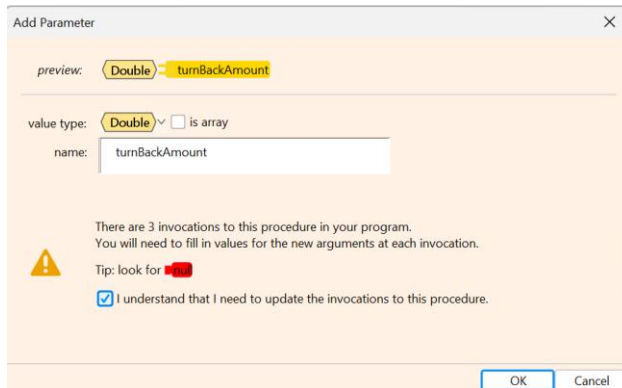
Add Parameter

Edit the wave procedure by adding a parameter to replace the variable turnback. It will also be a double, but with a slightly different name: **turnBackAmount**.



In the wave procedure, click Add Parameter, select **double** and name it **turnBackAmount**, the value that you give it doesn't matter because we will use the value that is passed as an argument.

We have already added some calls to the wave procedure in **myFirstMethod**, so we will get a message that we need to change the existing calls to the procedure. Check the box that tells us that we need to update the existing calls.



Note: Although the built-in procedures such as *say* have optional arguments, you cannot make these user defined parameters optional.

```
void myFirstMethod( )
```

do in order

```
// Yeti Waves
// Programmer: Janet Joy
```

```
ThreadUtilities.doTogether( ()-> {
    this.yeti wave ( null );
}, 0 -> {
    this.witch wave ( null );
}, 0 -> {
    this.troll wave ( null );
});
```

We need to add a value for **turnBackAmount** to each call. We have a value of 0.2 in the current code, so we can use 0.2, plus some slight variation such as 0.3 and 0.15

```
void myFirstMethod( )
```

do in order

```
// Yeti Waves
// Programmer: Janet Joy
```

```
ThreadUtilities.doTogether( ()-> {
    this.yeti wave ( 0.2 );
}, 0 -> {
    this.witch wave ( 0.15 );
}, 0 -> {
    this.troll wave ( 0.25 );
});
```

Use the Parameter

Next, we will edit the wave procedure to use the parameter instead of the variable **turnBack**, then we can delete the variable **turnBack**.

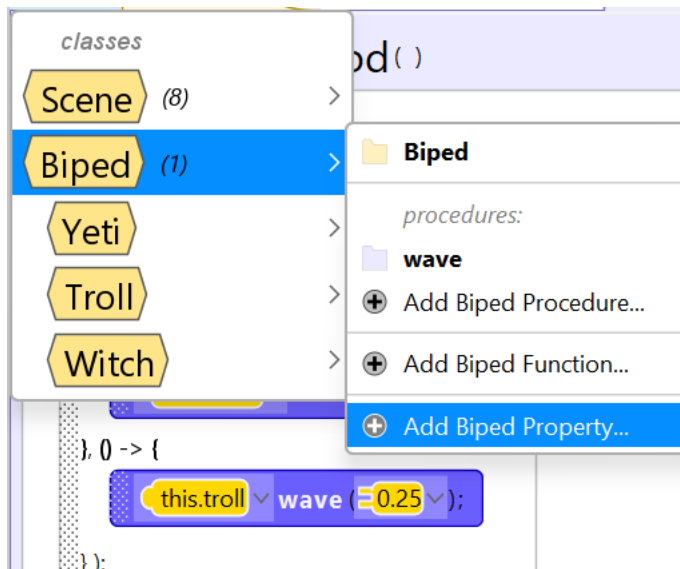
```
public void wave( Double turnBackAmount ) {
    // This is all of the code for waving
    // Get arm into position
    doTogether( () -> {
        this.say( "hello", Say.duration( 0.5 ) );
    }, () -> {
        this.getRightShoulder().turn( TurnDirection.LEFT, turnBackAmount );
    }, () -> {
        this.getRightElbow().turn( TurnDirection.BACKWARD, turnBackAmount );
    } );
    // Wave three times
    Double waving = 0.16;
    Double turnRight = 0.16;
    this.getRightElbow().turn( TurnDirection.LEFT, waving );
    for( Integer indexA = 0; indexA < 3; indexA++ ) {
        this.getRightElbow().turn( TurnDirection.RIGHT, turnRight );
        this.getRightElbow().turn( TurnDirection.LEFT, waving );
    }
    // Back to straight arm
    doTogether( () -> {
        this.say( "Bye" );
    }, () -> {
        this.getRightShoulder().turn( TurnDirection.RIGHT, turnBackAmount );
    }, () -> {
        this.getRightElbow().turn( TurnDirection.FORWARD, turnBackAmount );
    } );
}
```

Add More Parameters

We still have variables **waving** and **turnRight**, so we will replace them with variables also. We can go further: maybe we want to add strings for what they say instead of everyone saying “Hello” at the start and “Bye” at the end. The current code has everyone move their hand back and forth 3 times. We could use a variable for that also.

Add Biped Property

Another thing we can do is add a Biped Property. Select a biped, then go to **Biped** like before, but this time select **Add Biped Property**.



Add Biped Property

preview: **String** greeting = "hello"

is variable: ☒ variable field ☐ constant field

value type: **String** ☐ is array

name:

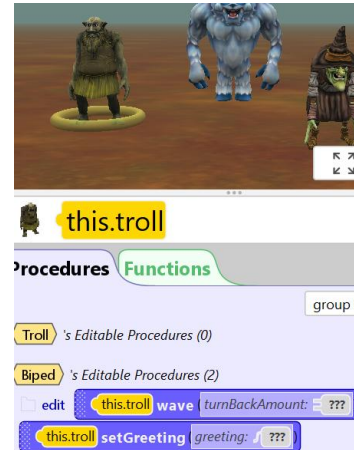
initializer: "hello"

We will call the new property **greeting** with a value of “hello”

Now, when you select a biped, you will see **setGreeting** as one of the commands. Drag the **troll.setGreeting** into the code and give it a value of “Howdy!”

Next, we will edit the wave procedure. Instead of just saying “Hello”, they will each say whatever their greeting property is. If we don’t set a value, they will say “Hello”

```
void myFirstMethod( )
do in order
  // Yeti Waves
  // Programmer: Janet Joy
  this.troll setGreeting ("Howdy!");
  ThreadUtilities.doTogether( ()-> {
    this.yeti wave (=0.2);
  }, 0 -> {
    this.witch wave (=0.15);
  }, 0 -> {
    this.troll wave (=0.25);
  });
};
```



Change wave to use Biped Property greeting

Instead of the string “hello”, we use **this.greeting**. Remember, **this** refers to whichever biped calls the procedure.

```
ThreadUtilities.doTogether( ()-> {
  this say (this.greeting, Say.duration(=0.5), add detail);
}, 0 -> {
  this.getRightShoulder 0 turn TurnDirection.LEFT, turnBackAmount
}
```

This statement in **myFirstMethod**: `this.yeti wave (=0.2);` means that in the **wave** procedure for that call, **this** refers to **yeti**.

When we run the program yeti says “**Howdy**”, the other two bipeds both say “**hello**”. (The value we gave the property when we created the biped property greeting.)

Other Class Procedures and Properties

These examples created a wave procedure and a property for the **Biped** class, but you can do the same for the other classes also.