CS 105 & CS106

Introduction to Computer Programming 1 & 2

Code Style Guide

Last revised: September 30, 2019

Introduction

The code you submit for labs and assignments is made more readable by paying attention to style. This includes using semicolons, declaring variables, whitespace, placement of comments, and choice of variable and function names. None of these affect the way the program executes, but they do affect the readability of your code.

Just like English, computer code is a language. The main goal of all languages is to facilitate communication. When writing code, you are not only communicating with a computer, but also with yourself (often your "future" self) and other people reading your code. In industry, those other people are typically other programmers. In this course, those other people will be your instructor and your TAs who are leading labs and marking your work.

This is why it is important to consider the needs of human readers and present code in a way that most clearly communicates what the program does. This is just like how we use indentations, commas, and periods to signify the start of a new paragraph, a pause in a sentence, or the end of a sentence.

Here is an example of the last two paragraphs without any common English language style rules (capital letters, commas, periods, breaks for paragraphs, etc.). Try reading it:

> just like english computer code is a language the main goal of all languages is to facilitate communication when writing code you are not only communicating with a computer but also with yourself often your future self and other people reading your code In industry those other people are typically other programmers in this course those other people will be your instructor and your tas who are leading labs and marking your work this is why it is important to consider the needs of human readers and present code in a way that most clearly communicates what the program does this is just like how we use indentations commas and periods to signify the start of a new paragraph a pause in a sentence or the end of a sentence

This is what it is like to read code that does not consider proper style.

This guide will outline the proper use of whitespace, comments, variable naming, code structure, and clarity and efficiency.

Semicolons

JavaScript uses a more permissive and relaxed language syntax than other common languages. One such difference is how JavaScript considers the use of a semicolon at the end of function calls and variable assignments as optional. If you leave the semicolons off, JavaScript will try to guess where they should go and then it invisibly adds them in. As you can imagine, this can cause problems. For this reason, the majority of the JavaScript community types semicolons to explicitly place them where they should go. In this course, we also require you to always type semicolons.

✗ *Leaving of Semicolons is Poor Style:*

```
let space = 5
function setup() {
  space = space * 2
  rect(0, 0, space, space)
}
```

✓ *Typing Semicolons Where They are Needed is Good Style*

```
let space = 5;
function setup() {
  space = space * 2;
  rect(0, 0, space, space);
}
```

Whitespace

When we talk about whitespace in computer programming, we are talking about the use of blank lines, indentations, and newlines to make our code more readable. To the computer, most whitespace is irrelevant, but it is very important for humans reading your code. This section presents guidelines for making good use of white space.

Let’s start with an example to illustrate poor use of whitespace and good use of whitespace.

✗ *Example of Poor Whitespace Style:*

```
let     space=5;
let number_of_circles=10;
let x=0;
function setup(){createCanvas (700,100) ;
colorMode (HSB,360,100,100,100) ;}
function draw (){background(360);
for( let i=0;i<number_of_circles;i ++ ) {
let hue=map(i,0,number_of_circles-1,150,250) ;
fill(hue, 80, 80);
let x2=x-i*(30+space); let y=height/2;
ellipse(x2,y,30,30) ; }
x+=2;
}
```

The JavaScript code above does not make use of blank lines, has no indentation, and is inconsistent with inline whitespace. Now, here is the same block of code using good whitespace style.

✓ *Example of Good Whitespace Style:*

```
let space = 5;
let number_of_circles = 10;
let x = 0;

function setup() {
   createCanvas(700, 100);
   colorMode(HSB, 360, 100, 100, 100);
}

function draw() {
  background(360);
  for (let i = 0; i < number_of_circles; i++) {
    let hue = map(i, 0, number_of_circles - 1, 150, 250);
    fill(hue, 80, 80);
    let x2 = x - i * (30 + space);
    let y = height / 2;
    ellipse(x2, y, 30, 30);
  }
  x += 2;
}
```

Basic In-line Spacing

One space after commas in function arguments:

✓ point(x, y);

✗ point(x,y);

No space between function name and opening bracket:

✓ ellipse(0, 10, 20, 30);

✗ ellipse (0, 10, 20, 30);

No space before the semicolon at the end of a statement:

✓ ellipse(0, 10, 20, 30);

✗ ellipse(0, 10, 20, 30) ;

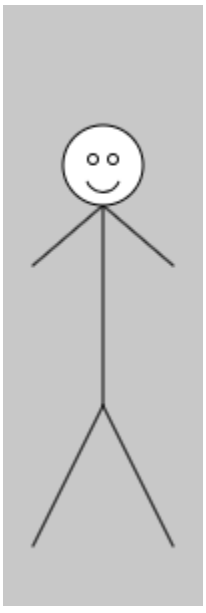
No space after the opening bracket or before the closing bracket:

✓ ellipse(0, 10, 20, 30);

✗ ellipse(0, 10, 20, 30);

Basic Use of Blank Lines

When discussing basic use of blank lines, consider a program to draw the stick figure at right:



✗ *Example of poor code style to draw this stick figure:*

```
createCanvas(100, 300);
background(200);
ellipse(50, 80, 40, 40);
ellipse(45, 77, 5, 5);
ellipse(55, 77, 5, 5);
arc(50, 85, 17, 17, 0.4, PI - 0.4);
line(50, 100, 50, 200);
line(50, 100, 15, 130);
line(50, 100, 85, 130);
line(50, 200, 15, 270);
line(50, 200, 85, 270);
```

Notice that it is hard to understand what this code does since there are no natural breaks. Let's now use blank lines and comments to make this easier to read.

Blank Lines

Blank lines inserted between groups of related statement provides visual structure to your code by dividing it into "chunk." Code is typically related by purpose in your algorithm

✓ *Example using blank lines to "chunk" the code into sections:*

```
createCanvas(100, 300);
background(200);

ellipse(50, 80, 40, 40);
ellipse(45, 77, 5, 5);
ellipse(55, 77, 5, 5);
arc(50, 85, 17, 17, 0.4, PI - 0.4);

line(50, 100, 50, 200);
line(50, 100, 15, 130);
line(50, 100, 85, 130);
line(50, 200, 15, 270);
line(50, 200, 85, 270);
```

This is a little better than the first version because now the reader understands that the chunks of code go together, but it still is not clear what all of these chunks of code do. The first chunk has only one statement which is creating the canvas, but all we can tell from the second third chunk at a glance is that they are probably drawing a part of the figure, but it isn't clear what part. That's where comments come in.

Comments

Comments provide information to the reader that cannot be understood easily by only reading the code. They may describe what a block of code does or explain why the code is needed. Comments may also be useful during the coding process.

Here is the previous code with comments to explain that the first chunk is setting up the canvas, the second chunk draws the head, and the third chunk draws the body.

Example using comments to explain chunks of code:

```
// Setup a tall grey canvas
createCanvas(100, 300);
background(200);

// Stick figure's head
ellipse(50, 80, 40, 40);
ellipse(45, 77, 5, 5);
ellipse(55, 77, 5, 5);
arc(50, 85, 17, 17, 0.4, PI - 0.4);
// Stick figure's body
line(50, 100, 50, 200);
line(50, 100, 15, 130);
line(50, 100, 85, 130);
line(50, 200, 15, 270);
line(50, 200, 85, 270);
```

This code is now much easier for another person to understand what this code is doing.

Using Comments

Comments most often appear on the lines before the code chunk or line of code that is being commented. There is no blank line between a comment line and the next line of code being described.

Whole line comments are on the line before the code they're commenting:

✓
```
// setup the canvas
createCanvas(100, 250);
```

✗
```
// setup the canvas
createCanvas(100, 300);
```

✗
```
createCanvas(100, 300);
// setup the canvas
```

Comments are occasionally placed on the same line as the code, always to the right of the code. Comments placed on the same line as code should always be a very short line of code and the comment itself should also be very short – at most 3 words. Comments for long lines of code or comments that require more detail should go on the line before the code being commented.

Same line comments are for short lines of code and short comments:

✓
```
ellipse(45, 45, 5, 5); // left eye
```
✗
```
ellipse(45, 45, 5, 5); // draw the left eye of the stick figure
```
✗
```
arc(50, 50, 20, 20, 0.4, PI - 0.4); / draw smiling mouth
```

Comments can have different purposes.

*Describe **what** the code does*

```
// check if car crashes into the curb
if (carX > curbRight || carX < curbLeft) {
...
```

*Describe **why** the code is needed*

```
// sometimes number is 0, which would be bad
if (number != 0) {
...
```

*Describe things that **may not be clear***

```
fill('#37BEC9'); // chartreuse
```

*Provide a **reference***

```
// Based on gravity formulas from:
// http://www.physics.com
function calculateGravity(heightFromGround, mass) {
...
```

Comments are sometimes used to make notes to yourself.

*Mark code with **TODO** when it needs to be completed*

```
// TODO: check if the car is outside canvas
```

*Mark code that is a **HACK** and should be cleaned up*

```
// HACK: need to parameterize this drawing
```

Comments to Avoid

Avoid restating what is already clear and obvious from the code

✗
```
//add b to a
a = a + b;
```

✗
```
// check if the left mouse button is pressed
if (mouseIsPressed && mouseButton === LEFT) {
…
```

✗
```
// declare the car width and set it to 100
let carWidth = 100;
```

Whitespace for Functions

When defining a function like setup, at least one blank line should come before the function. If you have a comment before the function then the blank line should come before the comment.

✓ *Example of good style with a blank line before function definition:*

```
function setup() {
  createCanvas(300, 300);
}

function draw() {
   ellipse(mouseX, mouseY, 10, 10);
}
```

✗ *Example of poor style with no blank line before function definition:*

```
function setup() {
  createCanvas(300, 300);
}
function draw() {
  ellipse(mouseX, mouseY, 10, 10);
}
```

Code Block Indentation

Anything between { and } is called a *code block*. All code and comments within a code block should be indented by the same amount using a single TAB or the same number of SPACES. Doing so allows the reader to easily see the code that belongs to each command that created the block. Some commands that create code blocks are while loops, for loops, built-in functions, and user defined functions.

Code blocks have the following format:

- The first line is the command. This line will always end with an open curly bracket '{'.
- The next lines are associated with the command are indented. The indent shows this association.
- The last line of a code block is a closing curly bracket '}'. It is not indented.

✓ *Good code block style with correct indenting:*

```
function setup() {
  createCanvas(150, 150);
  fill(200);
}
```

✗ *Poor code block style (no indenting):*

```
function setup() {
createCanvas(150, 150);
fill(200);
}
```

✗ *Poor code block style (incorrect indenting):*

```
function setup()
{
  createCanvas(150, 150);
  fill(200);
}
```

✗ *Poor code block style (incorrect indenting):*

```
function setup() {
  createCanvas(150, 150);
  fill(200);
  }
```

Advanced Indentation for Code Blocks within Code Blocks

It is possible to have a code block "nested" inside another code block. When this occurs the same rules described above are applied to both the outer and inner code blocks.

✓ *Good nested code block style with correct indenting:*

```
function draw() {
  background(200);
  strokeWeight(5);

  while (i < 10) {
    point(i * 10, 50);
    i++;
  }
}
```

In the example above, all code in the draw function is indented and within this code block there is a while loop. All the code in the while loop code block is indented again. Also, notice that the closing braces are indented according to what command it is associated with.

Advanced Indentation for Conditionals (if, else if, else)

The commands if, else if, and else follow the same indentation principles mentionedabove. The only difference is related to the closing bracket and the next command. An `if` statement is often followed by an else if statement or an else statement. Since these commands are logically linked together, the closing curly bracket and the next command occur on the same line.

✓ *Good indentation style for conditional statements:*

```
if (key === '0') {
  white = false;
} else if (key === '1') {
  white = true;
} else {
  white = !white;
}
```

✗ *Poor indentation style for conditional statements:*

```
if (key === '0') {
  white = false;
}
else if (key === '1') {
  white = true;
}
else {
  white = !white;
}
```

It is possible to have an if statement nested within another if statement. In these situations, the indenting is the same as described above.

Good indentation style for conditional statements:

```
if (i < 1) {
  if (i === 0) {
    print(“zero”);
  } else {
    print(“negative”);
  }
} else {
  print(“positive”);
}
```

Automatic Indentation in the Processing IDE using **CTRL+T**

A good strategy to instantly improve the indentation in your code is to press CTRL+T in the Processing IDE.

Note that this generally only fixes indentation, and it will sometimes not fix everything.

Advanced In-line Spacing

Operators are the things that do the action in equations or relational statements (e.g. the plus sign ‘+’ is an operator).

Binary Operators

Most operators are binary, meaning they operate on two values. Those values may be numbers, variables, or functions that return a value. Always add spaces around binary operators to make it easier to read the operators and values.

The assignment operator: =

✓
```
let circle_size = 10;
```

✗
```
let circle_size=10;
```

*Arithmetic operators: + - * / += -= *= /= %*

✓
```
let x2 = x - i * (size + space);
```

✗
```
let x2=x-i*(size+space);
```

Relational operators: === != = < <= > >=

✓ `if (x > width) {`

✗ `if (x>width) {`

Logical operators: && ||

✓ `if (x > width && keyIsPressed) {`

✗ `if (x>width&&keyIsPressed) {`

Unary Operators

Some operators are unary, meaning they operate on one value only. Do not add spaces between the unary operator and the value it operators on.

Increment operator ++ or decrement operator —

✓ `i++;`

✗ `i ++;`

Logical not operator !

✓ `if (!mouseIsPressed) {`

✗ `if (! mouseIsPressed) {`

Functions

Where you are calling a function or defining a function, there is never a space between the function name and the opening bracket.

Calling a function:

✓ `ellipse(0, 10, 20, 30);`

✗ `ellipse (0, 10, 20, 30);`

Defining a function:

✓ `function circle(x, y, size) { ...`

✗ `function circle (x, y, size) { ...`

Keywords

Keywords are commands that are part of the core language. Some P5 keywords have arguments and associated code blocks, a little like functions. However, standard coding style treats keywords different than functions: you leave a single space between the keyword and the opening bracket where the keyword "arguments" are. The only keywords you will see in this course that work like this are: if, while, for

Keyword inline spacing examples:

✓ `if (x != true) {...`

✗ `if(x != true) {...`

✓ `while (i < 10) { ...`

✗ `while(i < 10) { ...`

The three arguments in a for loop are separated by semicolons. These are treated exactly like commas in a function call.

Semicolons in for loops:

✗ `for (let i = 0;i < 10;i++) {...`

✓ `for (let i = 0; i < 10; i++) {...`

Breaking up Long Lines

Insert a line break for long lines of code. This lets you control how it wraps and ensure you can see the line in the code editor.

Example 1

✗ *Poor style due to very long line:*

```
  if (mouseIsPressed === true && mouseButton === RIGHT) || (keyIsPressed === 
true && key === 'x')) {
     print("right mouse button or x key!");
  }
```

✓ *Good style after wrapping the long line:*

```
  if (mouseIsPressed === true && mouseButton === RIGHT) ||
     (keyIsPressed === true && key === 'x')) {
     print("right mouse button or x key!");
  }
```

Example 2

✗ *Poor style due to very long line:*

```
rect(shipX - shipWidth/2 + offsetX, shipY - shipHeight/2 + offsetY, shipWidth,
shipHeight);
```

✓ *Good style after wrapping the long line:*

```
rect(shipX - shipWidth/2 + offsetX,
     shipY - shipHeight/2 + offsetY,
     shipWidth, shipHeight);
```

Variables

Declaring Variables

All variables should be declared using the let or const keywords. Do not use var

✗ *Poor style to declare new variables since no let or const:*

```
carPos = 100;
SIZE = 50;
```

✓ *Good style by using let or const:*

```
let carPos = 100;
const SIZE = 50;
```

Naming Variables

The way in which you name these variables also helps make your code more descriptive. Here are some guidelines for naming variables:

- Begin all variables with a lowercase letter
- One-character names should be avoided, except for temporary and looping variables.
- Adopt a consistent convention when your variable has two words like "lowerCamelCase": e.g.shipWidth, lastShipSpeed

When to Use Variables

Variables are used for values that are used over and over again. They are also used for values that are changed during the execution of a program.

Numeric values that appear in the code are often referred to as "magic numbers". A "magic number" is a number that over time may need to be changed or is repeated multiple times representing the same value. If we assign the "magic number" to a variable, then we can

update its value in one place if we need to, rather than multiple places. Magic numbers are a common source of errors in a program.

Consider an example with magic numbers for the size of the circles and their spacing.

Example with magic numbers:

```
function draw() {
  for (let i = 0; i < 10; i++) {
    ellipse(i * (10 + 50), 80, 50, 50);
  }
}
```

Example with variables substituted for magic numbers:

```
// so we can easily control the size and spacing
let circleSize = 50;
let circleSpacing = 10;

function draw() {
  for (let i = 0; i < 10; i++) {
    ellipse(i * (circleSpacing + circleSize), 80, circleSize, circleSize);
  }
}
```

Note that in the example above, we didn’t change every numeric value to a variable. If a value is only used once and it’s unlikely to change, then it doesn't need to be a variable.