

[the academy\_of\_code]

**Grade 9**

**Unit 1**

**Advanced Classes**

**Lesson Links:**

[Lesson 1 - Classes Recap](#)

[Lesson 2 - Inheritance](#)

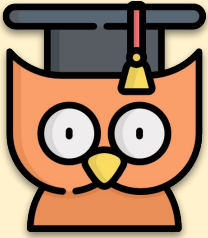
[Lesson 3 - Interfaces and Abstract](#)

[Lesson 4 - Making a GUI](#)

[Assessment](#)

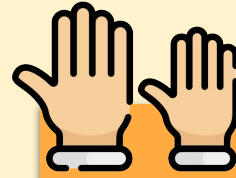
**www.theacademyofcode.com/handouts**

# Lesson 1 - Classes Recap



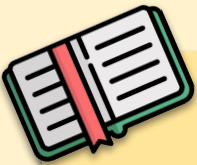
## Learning Outcomes:

- Recap class basics
- Refresh your knowledge of class terminology



## REMEMBER:

Put up your hand. We love to help!



## You've Seen Classes Before

The first lesson of this unit will be relatively short. Everything covered here is stuff you've either explicitly seen before, or have seen by proxy of what we've had you do in relation to classes in the past.

The purpose here is just to make sure you remember and understand everything you're supposed to before we explore the real reasons classes are used and so important in programming.

## Classes are Class

Knowing when and how to use classes really takes your programming ability to the next level. They are used to section our code into related chunks that describe some sort of *thing* in our code. They make the code we write easier to read, write and make changes to (called **maintaining** code).

Let's do a quick recap of some of the important aspects of classes and their terminology. We'll build a class while doing this to make things clearer and more interesting to follow.

## A “Book” Class

We’re going to create a class that represents a basic book, and explain all the jargon and processes as we go along.



### Planning our class

Before we write any class we should think about what it represents, and the kinds of fields it’s going to need. In our case, It’s going to represent a Book. What properties about a book differentiate it from other Books?



### Let’s Get Coding

- 1 Open a new sketch and in a new tab called “Book”, let’s declare a class called “Book”:

```
sketch_230126b  Book
class Book {
}
```

- 2 What are the properties that books have? Some basic ones are definitely a title, an author, the number of pages the book has, and even what page is currently open/what page the reader is currently at.

Let’s add variables (also known as **fields** in classes) to our class to store all this information.

```
class Book {
    String title, author;
    int pageCount, currentPage;
}
```

**Remember:** While this class has some simple fields, nearly any variable type can be a field in a class, including other types of classes!

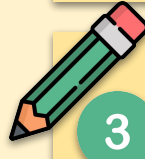


## The Constructor

Remember that a class is just a blueprint of something, it doesn't actually represent the physical thing itself. That won't exist until we *instantiate an instance* of our class.

*Instantiating* is what we do when we say something like "Book b = new Book();" in void setup(). The "Book()" part is us calling the constructor in the Book class.

The parameter list of a constructor can be empty. You can also choose to write an empty constructor, or omit it completely. If you do omit it yourself, java will add an empty one in behind the scenes so you're still able to instantiate objects.

- 
- 3 Let's add the constructor to the book class. When creating a book object, we need to specify the title, author and page count, but when we create a book the current page being read will always be 1.

```
class Book {  
    String title, author;  
    int pageCount, currentPage;  
  
    Book(String title, String author, int pageCount) {  
        this.title = title;  
        this.author = author;  
        this.pageCount = pageCount;  
        currentPage = 1;  
    }  
}
```



### this.

You'll notice that we use **this.** for the first three but not the last one. **this.** is used when we need to differentiate between the parameter *title* and the book field *title*. We are setting the current books title (this.title) equal to whatever the value of the passed in parameter title (title) is.

There is no parameter passed in called *currentPage* (there is no need since it will be 1 for every book we create) so there is no need to say *this.currentPage* as there is no other currentPage variable to mix it up with.

It's done this way because it's common java convention! In a weird counterintuitive sort of way, it also makes it easier to keep track of what variable is what.



- 4 Let's add some functions (also known as **methods**) to our book class that let us move to the next page, go to the previous page, and display what page we're currently on (along with the title and author of the book).

You need to add three functions:

- **read()**: Displays the title, author and currentPage on the screen.
- **nextPage()**: Updates the currentPage to move up 1 (as long as they haven't reached the end of the book!).
- **previousPage()**: Updates the currentPage to move down 1 (as long as they won't arrive on page 0!).

- 5 In your main tab, add in a void setup() and void draw(), create an instance of your book class and test out your functions. Use keyPressed() and your respective book functions for changing pages. Mine ended up looking like this:

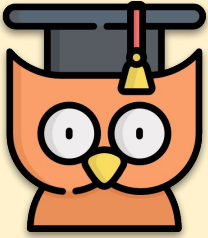


## Challenge Task

Create 2 more books and store all 3 of your books in an ArrayList. Use the number keys 1,2 and 3 to switch between which book you're currently reading and displaying on the screen.

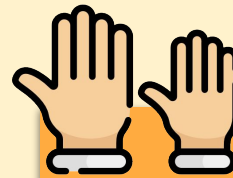
**Hint:** Use **get()** along with an int to run the functions on the correct book.

# Lesson 2 - Inheritance



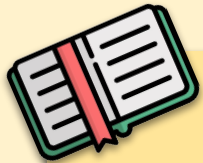
## Learning Outcomes:

- What is inheritance
- How do we use inheritance
- Where do we use inheritance



## REMEMBER:

**Put up your hand. We love to help!**



## What is Inheritance

In this lesson we are going to learn about inheritance. Inheritance is a mechanism in Java that allows one class to inherit (to get) attributes (variables) and methods (functions) from another class.

This allows us to more easily reuse code and allows us to structure our code in an manner that is easier to change, understand, and maintain. It also allows us to make effective use of polymorphism in our programs, making them ultra flexible.

## Uh... what?

If the section above was difficult to understand, that's expected. Now that you've reached Grade 9, a lot of the topics are going to be a bit hard to wrap your head around at first.

Remember, some people don't learn about this stuff until their second year of university, so you making it to this point is pretty impressive. Take a moment to be proud of yourself, seriously.

To make inheritance as a concept slightly easier to understand, let's run through an example of actually using it, which will let you see why it's super useful and pretty cool.

# Animals

Every animal is different. A dog and a cat look different, walk different, etc. But, they both sleep, they both breathe etc. The point is that while a dog and a cat are two different animals with different behaviours, at the end of the day they are both still an animal, and all animals sleep, breathe etc.

Previously, if we were to create classes for a dog and a cat in code, we would write the same functions for sleeping and breathing twice, one to go in the dog class and one to go in the cat class, even if the functions were exactly the same.

This is bad programming practice. We should try to remove all duplicate code if at all possible. This is where inheritance comes in to save the day. What we can now do instead is write a class *Animal* that contains methods for sleeping and breathing.

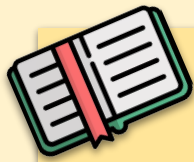
Then, we can write our dog and cat classes which will **inherit** from the animal class, allowing them to use the sleep and breathe functions that belong to the *Animal* class, without having those methods themselves.



## Let's Get Coding

- 1 Open up a new sketch and create a new tab called Animal with the following class in it:

```
class Animal {  
  void sleep() {  
    println("Sleeping");  
  }  
  void breathe() {  
    println("Breathing");  
  }  
}
```



## Super Class and Sub Class

What we just did in step one on the previous page was create what is known as the **Super** class (also called the **parent** class). In the next step we are going to create the class *Dog* that inherits the functionality of *Animal*, the super class. The *Dog* class is then known as a **sub** class (also called the **child** class) of the **super** class *Animal*.

All inheritance is made up of super and sub classes connected to one another, with sub classes inheriting (gaining) the functionality and variables of it's super class.



### Sub extends Super

- Let's create the *Dog* sub class. In a new tab called *Dog*, write the following code:

```
class Dog extends Animal {  
    void speak() {  
        println("Bark");  
    }  
}
```

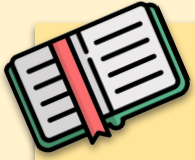
To let processing know that the class *Dog* is a sub class of the class *Animal* we use the keyword **extends**.

- Add the following code to your main program tab and run the program:

```
Dog dog;  
void setup() {  
    dog = new Dog();  
    dog.sleep();  
    dog.speak();  
}
```

The *Dog* class has no method **sleep()** but can run it as it inherits it from *Animal*. Notice how we also don't need to create an *Animal* object to inherit from it.





## Method Overriding

You've seen Method **Overloading** before, where we can have the same function that takes in different parameters called the same thing, that can perform differently depending on the parameters it receives.

Method **Overriding** is when a method in a sub class is called the same thing as a method in one of it's super classes. This can be used to give specific classes differing functionality for specific functions.



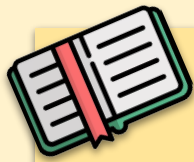
## Time to Eat

- 4 Add a new function to the *Animal* class called eating, that just prints "Nom Nom Nom".
- 5 In the *Dog* class, add a method called eating that prints "The dog is too giddy to eat!".

- 6 Change your main program tab to look like this and run it:

```
Animal animal;  
Dog dog;  
void setup() {  
    animal = new Animal();  
    dog = new Dog();  
    animal.eating();  
    println();  
    dog.eating();  
}
```

When the **eating()** function is called on the sub class *Dog*, it uses the **eating()** function contained within the *Dog* class as the **eating()** function in the super class has been overridden.



## super.

Sometimes, in a sub class, we want to perform some action unique to whatever class we're in, but also do some functionality contained within the super class alongside it.

You can access any variables/methods contained within the super class from the sub class by using **super.**(whatever variable/function you want).



- 7 In a new tab called Cat, create a new class *Cat* that is a sub class of *Animal*.

When the cat eats, we want it to print “Nom Nom Nom” but just before that, we also want it to print “The cat is hesitant but eats anyway.”

We can do this by **overriding** the eat function contained in animal, adding our own cat specific eating behaviour and then running the **()** function in the super class.

```
class Cat extends Animal {  
    void eating() {  
        println("The cat is hesitant but eats anyway.");  
        super.eating();  
    }  
}
```

- 8 Create a new Cat object and run *cat.eating()* at the end of void setup() and take a look at the output you get.

Please talk to your tutor about any questions you have about what you've just done, or things that you want to know that weren't already mentioned!





## SuperHero Fighting Game

You're going to create a text based game where two superheroes will fight to the death! To do this, we'll use a super class for the idea of a super hero, and then have sub classes for particular super heroes so that they can have slightly different stats and attacks.

- 1 In a new sketch, create a new tab called SuperHero, and in here create a *SuperHero* class. This class should have the following:
  - An **int**, health
  - A **string**, name
  - An **int**, attackDamage
  - A **constructor** that creates a hero based on a passed in name. The heroes health should be set to 100, and their attackDamage should be a random number between 1 and 20.
  - A **method** "isDead" that return true if the superheroes health is  $\leq 0$ , and false otherwise.
- 2 Add a method "attack" to the *SuperHero* class.
  - This method should take a *SuperHero* object as a parameter and subtract the attackers damage from the passed in SuperHero's health pool.
  - If the health of the attacked falls below 0, set it = to 0.
  - Print a line to the console explaining what happened, who just attacked who with how much damage and how much health is left etc.

```
void attack(SuperHero s) {
    s.health -= attackDamage;
    if (s.health <= 0) {
        s.health = 0;
    }
    println(name + " dealt " + attackDamage + " damage to " + s.name +
        ", leaving them with " + s.health + " health!");
}
```

- 3 Create two new SuperHero objects (called s1 and s2) in your main program and initialise them. Then copy the following for your void draw:

```
void draw() {  
    if (!s1.isDead() && !s2.isDead()) {  
        s1.attack(s2);  
        s2.attack(s1);  
  
        if (s1.isDead()) {  
            println(s1.name + " is dead!");  
        }  
        if (s2.isDead()) {  
            println(s2.name + " is dead!");  
        }  
    }  
}
```

Run your program and you should see your two superheroes battle to the death!

- 4 Our fights are a little bit boring right now, so let's spice them up by creating some custom heroes by using *inheritance*. Create a new class called "SuperMan" that is a subclass of *SuperHero* (it inherits from *SuperHero*).
- Superman should start with 150 health.
  - He should deal 25 damage, but has a 1/4th chance to take 25 damage instead of dealing damage (thanks to kryptonite!).

**Remember:** You can overload the attack function that belongs to the super class!

- 5 Create a new sub class of *SuperHero* called Batman
- Batman should have a variable "strength", that decreases by 2 everytime he attacks.
  - Batman should deal a random amount of damage each time he attacks, ranging from 2 damage to "strength" damage.
  - He should heal for 50% of the damage he deals each time he attacks.



## Challenge Task

Create 2 more custom superhero child classes. These can be existing superheroes you know of, or you can model one after yourself or just make one or both up entirely.

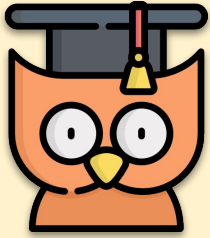
Be creative with how each different superhero works. Remember that the *SuperHero* class is the baseline to work with. You may want the attack moveset specified in the super class with a few additional bits added on, and you can just call **super.attack()** in your new superheroes attack function and then add on extra functionality you want.

You may also have an idea for a new mechanic that all superheroes should have, which we now know should go in our Super class so it can be inherited, instead of us having to write the same code 4 different times.

The goal here is to be flexible and creative while reusing all the code we can where possible. Ask your tutor for fresh ideas, or for help on how you can inherit specific functionality you need if you're a bit stuck.

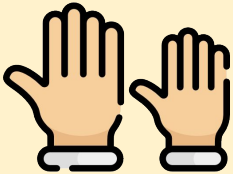
Also don't forget to take 5 minutes to balance your game a little bit!

# Lesson 3 - Interfaces and Abstract

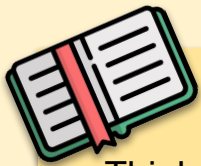


## Learning Outcomes:

- Understand Abstract Classes and Interfaces, and when and why they're useful



**REMEMBER:** If you have any questions, stay in your seat and put up your hand. We love to help!



## Let's Get Abstract

Think about the Animal example we used to learn how inheritance worked in lesson 2. In reality there is no such thing as just “an animal”. You can have a dog, or a cat, or a giraffe, or a mouse, they're all animals, but there is no animal that is just “an animal”.

That is all to say that the idea of an animal is an *abstract* idea. It has no concrete, real implementation. We saw in lesson 2 that we could create an object from the class animal, but we now know that this really doesn't make much sense.

What we should have done (and what we will do now in this lesson) is create an *abstract* animal class. A class in which we cannot instantiate (create an object from) but can have methods and be inherited from like other regular classes.

## Abstract Methods

An abstract class can have abstract methods. These are methods that don't contain any body in the abstract class, but must be defined and written in any non abstract class that inherits from the parent abstract class. We'll take a look at all of this using our animal example.



## Creating an Abstract Inheritance Structure

- 1 Let's create the animal class again, but this time we will define it as being an *abstract class* *Animal*. Give it a method called sleeping that prints three Z's.

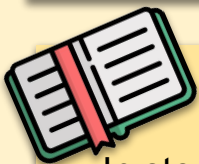
```
abstract class Animal {  
    void sleeping() {  
        println("Zzz...");  
    }  
}
```

In void setup(), try and instantiate an animal object.

```
Animal a;  
void setup() {  
    a = new Animal();  
}
```

Cannot instantiate the type sketch\_230130a.Animal

Our animal class is abstract, it is just an *idea* of something. Our program shouldn't be able to have just "an animal" like we discussed already. So by making our class *abstract*, we prevent anyone from creating an animal object.



## Concrete and Abstract Methods

In step 1 you saw we added a method to our class called **sleeping()**. This is because all animals sleep (that's semantically debatable but still), so it's fine to add in a method with functionality that all animals can inherit as it is the same for all of them. This is known as a concrete method.

What do we do when we know that all animals should be able to move, but they all move drastically differently (dog vs. fish vs. snake) so a concrete method wouldn't make sense? We can create an **abstract method**.





## Adding an Abstract Method

- 2 Add an abstract method called “move” to our animal class. To do this just add the keyword *abstract* at the beginning of the method declaration. This method will have no body (no implementation).

```
abstract class Animal {  
    void sleeping() {  
        println("Zzz...");  
    }  
  
    abstract void move();  
}
```

There is no “default” movement for an animal. Some animals have very similar movement yes, like all animals with 4 legs, but the class we’re dealing with is describing the highest level idea of an animal.

**Important:** Because this method is abstract, any (non-abstract) class that inherits from the Animal class **must** provide an implementation for the move function.

This is useful as we are now telling anyone that is creating a class for a specific kind of animal that it must be able to move and that they must be the one to provide the code to do it.

- 3 In a new tab, create a class *Dog* that inherits from our animal class. When you first create it, you’ll see an error for the Dog class telling you that you need to implement a *move()* function! Add the following code to your Dog class:

```
class Dog extends Animal {  
    void move() {  
        println("The dog sprints around");  
    }  
}
```



4 In void `setup()`, create an instance of your dog class and call the `sleeping()` and `move()` functions of the dog and see the output you get.

5 Create two new classes for any animal of your choice that inherit from the animal class. And write the move functions for these animals

6 Oh no! None of our animals have names! In the Animal class, add a new String variable called “name”. Add a constructor to your Animal class that takes in a String parameter and sets the name of the animal equal to the parameter.

```
abstract class Animal {  
    String name;  
    Animal (String name) {  
        this.name = name;  
    }  
}
```

7 In your other three classes you’ll see an error saying that they all require a constructor now. Add a constructor to each child class that takes in a string parameter, and then calls the super constructor with that passed parameter.



## Interfaces

Hopefully you can see how abstraction and abstract classes can be useful to keep other programmers in line, and even to help keep yourself on track. It’s nearly like planning in code, which is very handy.

Another way to achieve abstraction is to create and use an *Interface*. An interface is a completely abstract class with all methods having no bodies. While you *extend* a class, you *implement* an interface. You must implement all the functions defined in an interface when writing a class that implements the interface.

## Interfaces (Cont'd)

Interfaces are really useful for describing a specific thing in your program that must do certain tasks/things, but where you don't really care how it's done as long as the end goal reached is the same.



### Making and Using an Interface

Imagine you work for a big game development team. This team will soon start to add vehicles to the game they're making. Your job is to make sure that all the vehicles are standardised, that they all do the things necessary for them to be called "a vehicle".

How do you coordinate all the different developers? Do you go around 1 by 1 and tell them what methods their vehicles objects need to have? No.

The best way to do this is to create a vehicle interface. This way, once the other developers *implement* the interface, they know they have all the functionality required for their object to be considered a vehicle.

Let's do the basic version of this now.

1

In a new sketch, open a new tab and call it "Vehicle". Create a new interface called **Vehicle**:

```
interface Vehicle{  
  
}
```

Now, what defines a vehicle? Well, it needs to move, it needs to have passengers, it needs to refuel etc. These are the methods we should add to the interface.

- 2 Add in the methods we need to define a vehicle. Remember we're just sticking to the basics here so we're going to say that for something to be considered a vehicle, it must be able to move, store passengers, store cargo and refuel:

```
interface Vehicle{  
    void move();  
    void storePassengers();  
    void storeCargo();  
    void refuel();  
}
```

All these function are void, but they could be any other return type as well, depending on what the function is needed to do/what you want it to do to make sure your vehicle objects work well with the rest of your program.

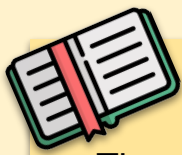
- 3 Create a new class called *Car*. This class is going to implement the Vehicle interface:

```
class Car implements Vehicle{  
}
```

There is currently an error on the car class. Remember that any class that implements an interface **must** implement all the methods outlined by the interface. There will be an error on the car class until all 4 methods have been written.

- 4 Write the methods for the car class. Occasionally it will move, pick up a passenger or cargo, or stop to refuel. To do this you can just print to the console.

You can trigger these events however you want, either randomly or using keyboard input etc.



## Why Bother?

The game development company you're working for comes back to you a few weeks later and says “**\*your name here\***, we actually want to see our vehicles move around on the screen! Make it happen!”.

We could just alter the car class, adding in a display method and altering our move method to watch the car move in the processing window, it would give the company what they want. But what if someone wants to make a new vehicle in the future?

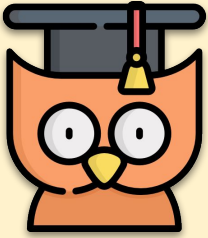
If they check the interface and use it correctly, they will likely break your program as when they try to run it and the *display()* method is called on their vehicle, well, they don't have one, so it will break.

But if you instead add to your interface, specifying that every vehicle needs a *display()* method, all our problems go away. Vehicles will not work at all until they have all the methods they need, preventing anybody writing some bad/incomplete code. This is why interfaces are useful.



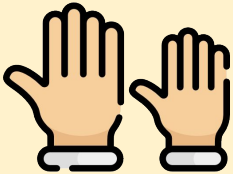
- 5 Add a new method to your interface, *display()*.
- 6 Write the implementation of this method in the car class and edit the move method so that when you run your program you see a car moving around the screen (bouncing off the walls) that you can refuel and pick up passengers/cargo with.
- 7 Add an *Airplane* vehicle. It should fly around at the top of the screen. Your passenger, cargo and refuel functions can be simple text again for this exercise.

# Lesson 4 - Anonymous Classes & Runnables

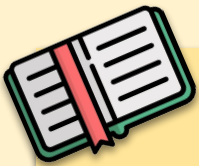


## Learning Outcomes:

- Learn how to make UI Buttons with Anonymous Classes and the Runnable class



**REMEMBER:** If you have any questions, stay in your seat and put up your hand. We love to help!



## Making a GUI with a Button Class

In the past, when you were making things like a menu screen or a settings menu, you probably coded up all the buttons, their collision detection and functionality separately. This was fine at the time as we didn't have too many buttons overall that we were worried about.

But now that we're becoming class at using classes, we can do it a much better way. We can instead make a new class, one that defines the blueprint of a button, containing all the functions needed for collision detection and drawing it on the screen etc.

Now all you'll need to do is create a new button object for each button you want, and we'll have repeatable, interactable buttons!

## Cookie Clicker

To demonstrate doing this, we're going to make the game Cookie Clicker, which I'm sure you've heard of. It's a good game to make when learning about this topic as the whole game is based on pressing/clicking buttons.

Let's get clicking!



## Making our Button Class

To start off, let's make a button class. Cookie clicker has multiple different buttons you can click with different icons that do different things. To match this, we're going to base our button class on `PImage`!

1 In a new sketch, open a new tab and create a class called *ImageButton*.

This class should have:

- A **PImage**: image
- **Ints**: xPos, yPos, w, h

2 The constructor declaration should look like the following:

```
ImageButton(String image, int xPos, int yPos, int w, int h) {
```

In the constructor, set all the variables to the given values, and load and resize the image.

3 Add a method called *drawButton* that will draw the image at the x and y position given for the button.

4 Finally, add another method called *detectCollision* that will perform rectangle collision detection with the mouse.

If a collision is detected, you should... wait... what goes inside the if statement? We're supposed to be able to use this class for every button in our game, but if all our buttons do different things, then won't the code inside the collision detection if statement need to be different every time?

Looks like we may have to just make a new different class for each button in our game, very tedious but some problems can't be solved.

Or can they...



## Anonymous Classes and the Runnable Interface

When we instantiate a button in `void setup()` (which we will get to), the problem is that we want to specify the functionality the button has when we create it. In other words, it would be great if we could pass in a button's functionality (what it does when we click it) as a variable.

This isn't possible in Java, but we can do *basically* the same thing using what's known as an **Anonymous Class**. This is a class with no name, that can be declared when needed. If we create an anonymous class that implements the [Runnable Interface](#), we will get a class we can declare with one method, `run()`, that contains the functionality we want the button to have.

Then we can just pass this class into our button constructor as a `Runnable` event, and run that event any time we click the button.

That likely didn't make a lot of sense, but we'll work through it as an example now so you can see it visually.

Also, this isn't *too* important to understand fully. It's more of a cool thing to show you that you will likely be able to use in future projects that need UI!



## Cookie Clicker (For Real This Time)

1

In this lesson we're focusing on using Anonymous Classes, so to get to a baseline where we can start, download this starter setup for the cookie clicker game.

Take your `ImageButton` class you created, and paste it into a new tab in the cookie clicker project under the same name. The project should be runnable (but do nothing) once this is done. If not talk to your tutor to see if you can get it fixed before moving on from this step.



- 2 Take a few minutes to read through all the code given in the main file to get to grips with how it all roughly works. There's just a few functions that draw shapes and text on the screen, and `void draw` is mostly concerned with calling those functions and drawing the buttons.

The *drawWarning* function will make more sense once your program starts running properly.

- 3 First things first, we need to update our *ImageButton* class. Give this class a new variable of type **Runnable** called "event":

```
Runnable event;
```

Add to the constructor declaration so that it also accepts a **Runnable** event parameter:

```
ImageButton(String image, int xPos, int yPos, int w, int h, Runnable event) {
```

Don't forget to assign the class's "event" variable to the value of the parameter "event"!

- 4 Let's create a button now in `void setup()`. We'll start with the easiest one, the main cookie, most of this is simple, except the last parameter, the **Runnable** event:

```
imageButtons.add(new ImageButton("cookie.png", 64, 86, 188, 188, ?);
```

We'll deal with the `?` next by replacing it with a **Runnable** object.



You might be wondering how we can create a **Runnable** object if **Runnable** is an interface. **Runnable** is one of the exceptions to the rule, as it is a functional interface that be used in things like lambda expression and method references, but you don't need to worry about this.



- 5 For the Runnable event parameter, we're going to create a new **Runnable** object that has one method, *run()*. *run()* in this case should just increase the number of cookies we have by 1, as that's what we want collision detection on this button to do:

```
imageButtons.add(new ImageButton("cookie.png",
    64, 86, 188, 188, new Runnable() {
        void run() {
            cookies++;
        }
    }
));
```

**Note:** This could all be done on one line, but it can be easier to read and understand if you break it into multiple lines like in the picture above.

- 6 For anything to happen when we click this button, we need to finish the *ImageButton* classes *detectCollision()* method.

For any particular button, we know that when we click that button, we want whatever functionality we wrote in that button's *run()* method in it's **Runnable** event to happen. So to do this, we can just say *event.run()*!

```
void detectCollision() {
    if (mouseX > xPos && mouseX < xPos+w &&
        mouseY > yPos && mouseY < yPos+h) {
        event.run();
    }
}
```

If you run your game now, your cookie count should increase by 1 every time you click the cookie! This is a lot of work for one button, but now we can add as many as we want, and to get different functionality, only the contents of *run()* needs to change instead of having to create a new class!

**7** Now that our framework is setup, we can just add in new buttons that should work. One of the basic items in cookie clicker is the cursor that adds increases our idle cookiesPerSecond by 1. Let's add this in (you have the cursor image from the starter file). The *run()* method for this button will look like this:

```
if (cookies >= cursorPrice) {  
  cookies -= cursorPrice;  
  cursorPrice *= 1.1;  
  cursors++;  
  cookiesPerSecond += 1;  
} else {  
  warning = "Not enough cookies for: Cursor.";  
  warningTimer = 5 * 60;  
}
```

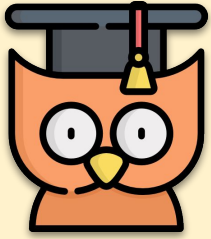
If you don't have enough cookies to afford a cursor, we will display the warning for 5 seconds and then make it disappear, otherwise we will subtract the price (which you should set to 10 at the start) from your cookies total, increase the price to 110% of its current price, add a cursor and increase your cookiesPerSecond by 1.

**8** **Add in 4 more buttons**, one using the grandma image, with the other 3 being whatever you want. You may need to do some resizing/rearranging of other things to make it look nice (maybe increase your screen width to fit 3 more buttons on the right side).

The grandma button should work the same way as your cursor button but should cost 50 to start with and increase your cookiesPerSecond by 10.

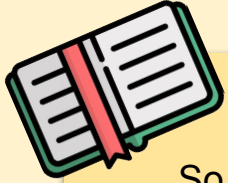
The other buttons can be whatever you want.

# Assessment



## Learning Outcomes:

- Be able to use Inheritance when needed to write better code
- Know how and when to use abstract classes/interfaces



## Attention!

So far in your coding journey with us, you will have completed assessments for Grades 4, 5, 6 and 7. These were essentially small projects where you were told exactly what to do and given a time limit to do it, with limited to no help from your tutor.

This assessment will not be given a strict time limit, and you can ask for help like you normally would, if you need it. On the next page you will find the criteria for this project. While there is no definitive marking scheme, there are certain objectives you will need to fulfill for your project to be considered finished.



## The Task

Remember the Bug Catcher game you made back at the end of Grade 7? There were a decent number of different bugs that, let's be honest, at the time we didn't deal with in a great way. This assessment requires you to build bug catcher (or a game of your choosing like it), but using what we've learned in this unit to make it "properly".

## Bug Catcher (Or Equivalent)

Remember that while these are technically a list of requirements, you're at a stage now in grade 9 where you can talk to your tutor and discuss an alternate idea you have.

As long as it's the same level of work and complexity (and you're making use of the important concepts covered in this unit) it should be fine to run with your own idea!

### Basic Requirements:

- A **character interface** that your player character and enemy *super class* (see next point) should inherit from.
- A player class (that implements the character interface) that should be able to move left and right and be seen on the screen.
- Your enemy (that implements the character interface) should be described using an *abstract class*, that may have none, one, or more than one concrete method(s).
- You should have **at least 4 different enemy archetypes** that inherit from your abstract enemy superclass, all with their own unique variations (like a super fast enemy, a bigger enemy, an enemy that helps you in one way and is a detriment in another way etc. Look back at the bug catcher lesson for some basic ideas.)
- Collision detection between the player and the enemies, (each collision will result in different things happening just like in bug catcher, but this will be easily written in void draw as long as your enemy classes are written well).
- Some form of win/lose condition

When you've finished the basic game, go to the next page to see how to polish and finish it.

## Game Polish and Extras:

- Add in a UI to your game. This would include a main menu and game over screen. There should be a settings menu where you can adjust some game parameters (see next point) and a menu button available to click while playing the game.  
Make a button class and use anonymous classes and the runnable interface to make your UI.
- The settings menu should let you change some game parameters. Here are some ideas:
  - Change rate at which enemies appear from the top of the screen
  - Change the players move speed
  - Change difficulty (by raising points needed to win/lowering lives etc.)
- Add a new enemy sub class to one of your existing enemy implementations so that it behaves and looks exactly like that enemy (you decide which one). This new enemy should be one that spawns in very rarely. If you hit this enemy, the whole program should just exit.