# Let's Remove Duplicate Code

This exercise shows how to **apply polymorphism** and **encapsulation** to create clean, reusable code.

**Polymorphism is really important.** If you want to master OOP and be able to build great apps using reusable software, you must first master polymorphism.

## The Problem

In Lab1, there's a lot of **identical code** in the tasks:

```
# Task 1
def sum_integers(max_value: int):
    print(f"Summing integers from 1 to {max_value:,d}")
    stopwatch = Stopwatch()
    stopwatch.start()
    sum = 0
    for n in range(1,max_value+1):
        sum += n
    stopwatch.stop()
    print("Finished")
    print(f"The result is {sum:,d}")
    print(f"Elapsed time: {1000*stopwatch.elapsed:.6f} millisec.")

# Task 3
def sum_decimal_fractions(max_value: int):
    print(f"Summing Harmonic series from 1 to 1/{max_value:,d}")
    stopwatch = Stopwatch()
    stopwatch.start()
    sum = 0
    for n in range(1,max_value+1):
        # add 1/n to the sum
    stopwatch.stop()
    print("Finished")
    print(f"The result is {sum:,.12f}")
    print(f"Elapsed time: {1000*stopwatch.elapsed:.6f} millisec.")

# and so on...
```

This is a lot of **duplicate code**. To add a new task, we need to copy and edit several lines of code.

If we want to **change** something like the print( ) statement, we need to change it at every occurrence.

Now that we see the problem, **let's remove the duplicate code!**

**As a bonus, the result will be reusable code.**

## How to Eliminate Duplicate Code?

To eliminate the duplicate we need to **separate the code that varies** from the **code that stays the same** for each task.

Code that Varies                                                    Code that Stays the Same for Each Task

```
# called from main
sum_integers(1000000)
                                    print(
# describe this task
f"Summing integers from 1 to {max_value:,d}"                          )
                                    stopwatch = Stopwatch()
                                    stopwatch.start()
sum = 0
for n in range(1,max_value+1):
    sum += n
                                    stopwatch.stop()
                                    print("Finished")
                                    print(f"The result is",
                  "{sum:,d}"                                          )
                                    millisec = 1000*stopwatch.elapsed
                                    print(f"Elapsed time: {millisec:.6f} ms")
```

To use polymorphism, make the code that varies ***look the same***: Encapsulate each task in an object with the same interface:

Code that Varies (Encapsulated)                                     Code that Stays the Same

```
class Task1:                        def tasktimer(task):
  def __init__(self, size):             # describe this task
      self.limit = size                 print("Task:", str(task))
                                        stopwatch = Stopwatch()
                                        stopwatch.start()
                                        result = task.run()
                                        stopwatch.stop()
                                        print("Finished")
                                        print("The result is ", result)
                                        millisec = 1000*stopwatch.elapsed
                                        print(f"Elapsed time: {millisec:.6f} ms.")
  def __str__(self):
      return f"Sum integers from 1 ..."

  def run():
      sum = 0
      # ... sum the numbers
      return sum
```

Each task is ***encapsulated*** in a runnable object.

Each task has the same public methods (same interface) so that tasktimer can use them in exactly the same way.

Now, the Code That Stays the Same does **not depend** on the Code that Varies **--** it only depends on the ***interface*** **(the methods)**, which is the same for every task.

## Assignment

1. Put each task in its own class

2. Write the tasktimer(task) function.

3. Test your code.

4. Go A Step Further - remove duplicate code in the "main" block
Your main block might now look like this:

```
tasktimer(Task1(1000000))
tasktimer(Task2(1000000))
tasktimer(Task3(10000))
// and so on...
```

All those calls to `tasktimer` are duplicate code, too.

Can you simplify the code even more?

```
tasks = [Task1(1000000), Task2(1000000), Task3(1000000), ...]
for task in tasks:
    # do what?
```

## Use a Task Superclass for Tasks?

If the tasks contain duplicate code (the constructor looks the same), you can create a superclass for Tasks and put the common code in the superclass.

## Credits

This exercise was created by Thai Pangsakulyanont.