

## Introduction to Software Engineering (ISAD1000)

# Lecture 7: Modularity

---

Updated: 7<sup>th</sup> May, 2020

Discipline of Computing

School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright © 2020, Curtin University

CRICOS Provide Code: 00301J

Design  
oooo

Maintenance  
oooo

Coupling  
oooooooooooooooo

Cohesion  
oooooooooo

Redundancy  
oooooooooo

# Outline

Design

Maintenance

Coupling

Cohesion

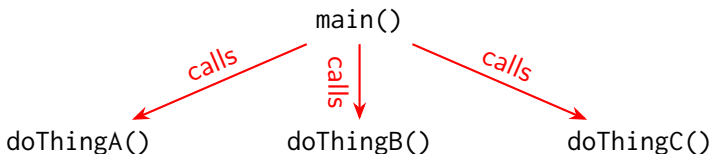
Redundancy

# Design

- ▶ Design is a half-way step between requirements and coding.
- ▶ Uses many notations:
  - ▶ Pseudocode,
  - ▶ Structural diagrams,
  - ▶ Behavioural diagrams,
  - ▶ Tables.
- ▶ However, it also lives inside your code!
- ▶ Design is the *set of ideas* you have about how to satisfy the requirements.
- ▶ Some of these are big picture ideas; some are small details.

## Modularity

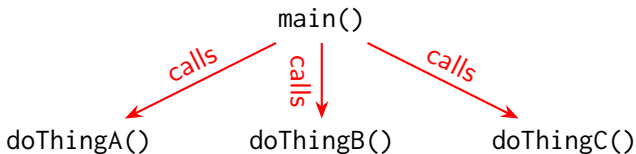
- ▶ A large part of design is about breaking things down.
  - ▶ What should the *parts* of your application be?
  - ▶ Divide and conquer – smaller problems are easier to overcome.
- ▶ We aim for *modularity*.
- ▶ Break up the software into self-contained pieces: methods, functions (and larger structures like “classes” and “packages”).
  - ▶ A “module” is a specific Python concept.
  - ▶ *But*, more abstractly, it means any sub-part of a program.
- ▶ These pieces (modules) use one another:



## Module Relationships/Dependencies

- ▶ Modules use each other to help accomplish tasks.
- ▶ Thus, modules depend on each other.
- ▶ Modules should hide their internal workings.
  - ▶ One module shouldn't need to “know” how other modules work.
  - ▶ More precisely, when writing/modifying a module, *you* shouldn't need to know how other modules work.
- ▶ It may not be obvious *why* this separation is a good idea. . .

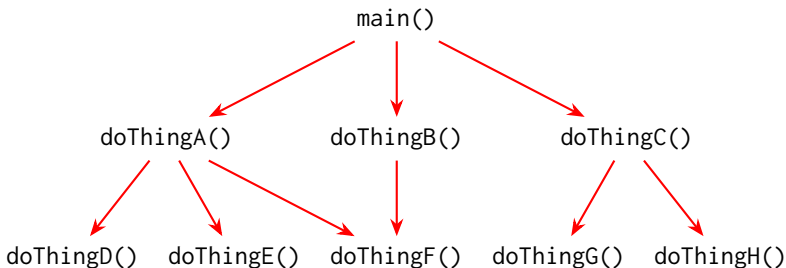
## What is the Problem?



- ▶ Is it important to have these methods/functions at all?
- ▶ Is it important to give them well-defined responsibilities?

## What is the Problem?

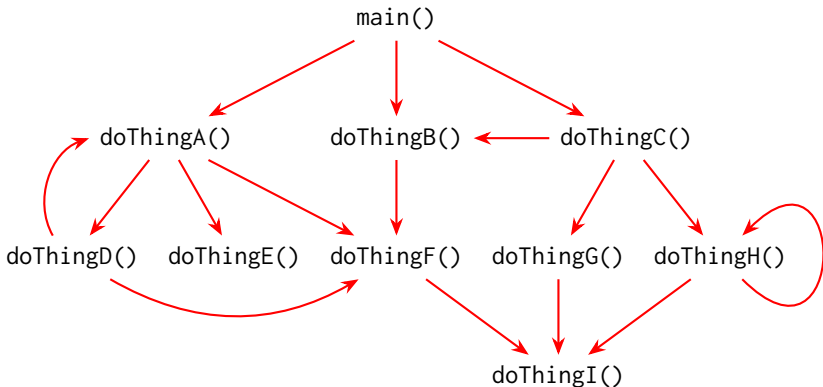
Programs can get larger:



- ▶ Painful to work with if you don't "divide and conquer" properly.
- ▶ So, ensure each method/function has a *single, well-defined responsibility*.

## What is the Problem?

And programs can get more complex:



- When the relationships are complex anyway, you need all the “simplification skills” you can get!



# Maintenance

- ▶ The final stage in the life of a software project.
- ▶ Occurs *after* the software is released or delivered to the client.
- ▶ The word “maintenance” is slightly misleading:
  - ▶ Hardware maintenance means fixing/replacing parts that become faulty or damaged over time.
  - ▶ This doesn't happen to software, which is just information.
  - ▶ If you find a software fault, that fault was *always there* (since at least the last modification, and often long before).
- ▶ There are good reasons to perform maintenance:
  - ▶ Corrective maintenance – fixing faults.
  - ▶ Perfective maintenance – improving or adding functionality.
  - ▶ Adaptive maintenance – updating the software for changing circumstances.

## Lehman's Laws

- ▶ Meir Lehman proposed 5 laws, based on observations of software projects.
  - ▶ These are “laws” in the scientific sense.
  - ▶ They are a statement of the way things are, *not* guidelines for how things should be.
- ▶ We'll focus on the first two:
  1. **Continuing change** – a useful program either undergoes continual change/evolution, or becomes progressively less useful.
  2. **Increasing complexity** – as a program changes/evolves, its design complexity increases and its structure deteriorates, unless extra work is done to compensate.
- ▶ The other three laws relate to the ability to measure and predict the course of a large software project, independently of the actual work that needs to be done.

## Refactoring

- ▶ Modifying your code without changing its functionality.
- ▶ Why?
  - ▶ To improve maintainability, and so counteract Lehman's 2nd law (increasing complexity).
  - ▶ To increase design flexibility, paving the way for future functionality to be added.
- ▶ Refactoring involves some redesign work.
  - ▶ It's not just about adjusting spacing, variable names, etc.
  - ▶ You choose a different, more *elegant*, more *logical* design.
- ▶ Often means:
  - ▶ Splitting up modules.
  - ▶ Combining separate modules into one.
  - ▶ Moving parts of one module into another.
  - ▶ Changing the way two modules communicate.
  - ▶ Eliminating redundant code.

## Regression Testing

- ▶ Modifying working code always carries a risk.
- ▶ You could introduce a new fault.
  - ▶ Called a *regression*.
  - ▶ Your program regresses from working to faulty.
- ▶ *Regression testing* checks whether this has happened.
  - ▶ Test-Driven Development makes it easy.
  - ▶ Most of your test code should still work with the modified production code.
  - ▶ Update any out-of-date test code.
  - ▶ Run the tests.
  - ▶ If anything breaks that was previously working, you have a regression.
  - ▶ Fix it!
- ▶ Now we'll get back to design.

## Coupling

- ▶ In FOP/PDI/OOPD, you learn that programs are broken down into several methods/functions.
  - ▶ In fact, even small programs have dozens of methods/functions.
  - ▶ Large programs have thousands.
- ▶ Methods/functions (and larger structures) interact in various ways, most obviously by “calling” each other. e.g.
  - ▶ `calcDaysInMonth()` must know if a given year is a leap year.
  - ▶ `calcDaysInMonth()` calls `isLeapYear()` to find out.
  - ▶ Thus, `calcDaysInMonth()` *depends* on `isLeapYear()`.
  - ▶ (`isLeapYear()` might also be called elsewhere in the same program.)
- ▶ But this isn't the only way that methods/functions can interact.

## Degree of Coupling

- ▶ Not all coupling is equal.
  - ▶ Some coupling is looser/lower.
  - ▶ Some coupling is tighter/higher.
- ▶ We prefer it to be as loose as possible.
  - ▶ The loosest coupling is no coupling at all.
  - ▶ However, *some* coupling is essential, or the program will become logically impossible to write.
- ▶ Where high coupling exists between two modules, the contents of one have very significant effects on the other.
  - ▶ Working with tightly-coupled modules is difficult.
  - ▶ You must understand both at once, rather than one-at-a-time.
  - ▶ This makes it more time consuming (and expensive) to write, test, inspect or modify the code.
  - ▶ So, avoid high coupling!

## Calls

- ▶ Calls are the most obvious and common form of coupling.
- ▶ A call (a.k.a an *invocation*) is a very specific event.
- ▶ e.g. when `calcDaysInMonth()` calls `isLeapYear()`:
  1. `calcDaysInMonth()` pauses.
  2. The year is passed from `calcDaysInMonth()` to `isLeapYear()`.
  3. `isLeapYear()` performs its calculation.
  4. The result is returned back to `calcDaysInMonth()`.
  5. `calcDaysInMonth()` resumes from where it left off.
  6. `calcDaysInMonth()` receives the return value and uses it in its own calculations.
- ▶ Parameters and return values makes the coupling slightly higher (than not having them).
  - ▶ We generally can't avoid this (without doing something much worse).
  - ▶ But it is possible to have *too many* parameters.
  - ▶ More than about 6 is a warning sign.

## Global Variables (or Public Fields)

- ▶ Normal (“local”) variables only exist within a particular method/function.
- ▶ Global variables exist outside any method/function.
- ▶ They can be accessed directly from anywhere.
- ▶ Lazy programmers use them as a short-cut.
  - ▶ “How can data get from doThingA() to doThingB()?”
  - ▶ “Ah ha! A global variable!”
  - ▶ Yes, but you will live to regret it.
- ▶ Global variables create tight (high) coupling between modules.
- ▶ The modules don’t even refer to each other, making the coupling very difficult to see.
  - ▶ But it’s there. Changes made to one module, in terms of how it uses the global variable, will affect all other modules that use it.



## Global Variable Example


```
public class GlobalVariableExample
{
    public int x;          // Global variables (or technically
    public int xSquared;   // "public fields" in Java).

    public static void main(String[] args)
    {
        x = ...; // Input a value from the user
        square();
        outputResult();
    }
    public static void square() {
        xSquared = x * x;
    }
    public static void outputResult() {
        System.out.println(xSquared);
    }
}
```



## Global Variable Example

```
def square():  
    global x  
    global xSquared  
    xSquared = x * x  
  
def outputResult():  
    global xSquared  
    print(xSquared)  
  
if __name__ == "__main__":  
    x = int(input())  
    square()  
    outputResult()
```

 **Python**

# Note: "global x" allows a function  
# to modify global variable x. You  
# technically don't need it simply  
# to read a global variable.

# Having told you how to do this...  
# don't!

## Global Variable Discussion

- ▶ In the previous example:
  - ▶ The main/top-level code is coupled to `square()` via the global variable `x`.
  - ▶ `square()` and `outputResult()` are coupled via the global variable `xSquared`.
- ▶ What's wrong with this?
- ▶ Problems arise when we want to *modify* the code (maybe to extend the functionality).
- ▶ Global variables are a minefield.
  - ▶ It's easier to make mistakes, and harder to fix them.
  - ▶ You can't easily see the consequences of what you're about to do, because the global variables connect things without telling you that they're connected.

## Global Variables Increase Complexity!

- ▶ Say we want to **square two numbers and add them**.
- ▶ *With* global variables:

```
x = ...; // Input 1st value
square();
int result1 = xSquared;
x = ...; // Input 2nd value
square();
int result2 = xSquared;
int result = result1 + result2;
```



- ▶ *Without* global variables:

```
int x = ...; // Input 1st value
int y = ...; // Input 2nd value
result = square(x) + square(y);
```



## Global Variables Increase Complexity!

- ▶ Say we want to **square two numbers and add them**.
- ▶ *With* global variables:

```
x = ... # Input 1st value
square()
result1 = xSquared
x = ... # Input 2nd value
square()
result2 = xSquared
result = result1 + result2
```



- ▶ *Without* global variables:

```
x = ... # Input 1st value
y = ... # Input 2nd value
result = square(x) + square(y)
```



## Global Variable Are Messy

- ▶ If x is global, then we can't do this:

```
x = ...  
y = ...  
square() # Both calls to square() will use x, and not y.  
square()
```

- ▶ We can't fix it like this either:

```
x = ...  
x = ... # This just overwrites the first value of x.  
square()  
square()
```

- ▶ A similar problem applies to the xSquared variable.

## Removing Global Variables

- ▶ Global variables can be removed by converting them into:
  - ▶ Parameters, when a method/function needs to import information;
  - ▶ Return values, where a method/function needs to export information;
  - ▶ Or both (if it was both reading and modifying a single global variable).

## Removing Global Variables

```
public class NoMoreGlobalVariables
{
    public int x;
    public int xSquared;
    public static void main(String[] args)
    {
        int x = ...; // Input a value from the user
        int xSquared = square(x);
        outputResult(xSquared);
    }
    public static int square(int x) {
        return x * x;
    }
    public static void outputResult(xSquared) {
        System.out.println(xSquared);
    }
}
```





## Removing Global Variables




```
def square(x):  
    global x  
    global xSquared  
    return x * x  
  
def outputResult(xSquared):  
    global xSquared  
    print(xSquared)  
  
if __name__ == "__main__":  
    x = int(input())  
    xSquared = square(x)  
    outputResult(xSquared)
```

## Control Flags

- ▶ Parameters are supposed to provide data needed to perform an operation.
- ▶ Sometimes, a parameter is nothing but a way of choosing between different operations – a control flag.
  - ▶ Often a boolean, but could actually have any type.
- ▶ When this happens, the *caller* method/function and *called* method/function are more tightly coupled than usual.
- ▶ The caller is not just using the called, but some *subcomponent* of the called.
- ▶ The caller depends (at least partly) on the inner workings of the called.

## Control Flags – Example

```
public static String formatDate(int one, int two,  Java  
                                int three, boolean isDate) {  
    String s;  
    if(isDate) {  
        s = one + "/" + two + "/" + three;  
    }  
    else {  
        s = one + ":" + two + ":" + three;  
    }  
    return s  
}  
  
public static void printDate() {  
    int day, month, year;  
    ... // Input values for day, month, year  
    System.out.println(formatDate(day, month, year, true));  
}
```

## Control Flags – Example


```
def formatTimeDate(one, two, three, isDate):  
    if isDate:  
        s = str(one) + "/" + str(two) + "/" + str(three)  
  
    else:  
        s = str(one) + ":" + str(two) + ":" + str(three)  
  
    return s  
  
def printDate():  
    ... # Input values for day, month, year  
    print(formatTimeDate(day, month, year, True))
```



## Control Flags – Discussion

- ▶ In the previous example, `formatTimeDate()` has a control flag parameter `isDate`.
  - ▶ If `isDate` is true, we format a date.
  - ▶ If `isDate` is false, we format a time.
  - ▶ `isDate` itself is not really data. It has no purpose other than to join together two unconnected tasks.
- ▶ `printDate()` really depends on *one half* of `formatTimeDate()`.
  - ▶ This is actually a tighter coupling arrangement, because `printDate()` has to “know” about time formatting, even though that’s not needed.
- ▶ A better solution would be to:
  - ▶ Split `formatTimeDate()` into `formatTime()` and `formatDate()`.
  - ▶ Have `printDate()` call only `formatDate()`.
  - ▶ Thus, eliminate the control flag altogether.

## Refactoring Control Flags

```
public static String formatTime(int hr, int min, int sec)  Java
{
    return hr + ":" + min + ":" + sec;
}

public static String formatDate(int day, int month, int year)
{
    return day + "/" + month + "/" + year;
}

public static void printDate()
{
    int day, month, year;
    ... // Input values for day, month, year
    System.out.println(formatTimeDate(day, month, year, true)
                      formatDate(day, month, year));
}
```

# Refactoring Control Flags



```
def formatTime(hr, minute, sec):  
    return str(hr) + ":" + str(minute) + ":" + str(sec)  
  
def formatDate(day, month, year):  
    return str(day) + "/" + str(month) + "/" + str(year)  
  
def printDate():  
    ... # Input values for day, month, year  
    print(formatTimeDate(day, month, year, True)  
          formatDate(day, month, year))
```

## Cohesion

- ▶ *Cohesion* is the extent to which a single module does one well-defined task.
- ▶ We want to *maximise* cohesion (just as we want to minimise coupling).
- ▶ High cohesion leads to more efficient use of your mental resources.
  - ▶ If a module has one well-defined purpose, it will be easier to understand.
  - ▶ If it's easier to understand, it will be faster to write, test, inspect and modify.




## Coupling vs. Cohesion

- ▶ Good (low/loose) coupling and good (high) cohesion go hand-in-hand.
- ▶ Good coupling and cohesion are facets of modularity.
- ▶ Improve one, and you often improve the other as well.
- ▶ If one is bad, the other tends to be bad as well.
- ▶ How to tell the difference?
  - ▶ Cohesion deals with tasks done *within a single* module.
  - ▶ Coupling deals with connections *between two* modules.
  - ▶ “Couple” literally means two – that’s how you remember which is which.

## Control Flags (Again)

- ▶ Control flags may also indicate low cohesion – where a method/function performs more than one distinct task.


```
public static String formatTimeDate(int one, int two,  Java  
                                   int three, boolean isDate) {  
    String s;  
    if(isDate) {  
        s = one + "/" + two + "/" + three;  
    }  
    else {  
        s = one + ":" + two + ":" + three;  
    }  
    return s  
}
```

- ▶ formatTimeDate() formats dates, and formats times.
- ▶ These tasks are similar, but *not really* a single responsibility.

## Control Flags (Again)

- ▶ Control flags may also indicate low cohesion – where a method/function performs more than one distinct task.


```
def formatTimeDate(one, two, three, isDate):  
    if isDate:  
        s = str(one) + "/" + str(two) + "/" + str(three)  
  
    else:  
        s = str(one) + ":" + str(two) + ":" + str(three)  
  
    return s
```



- ▶ formatTimeDate() formats dates, and formats times.
- ▶ These tasks are similar, but *not really* a single responsibility.

## Sequential Tasks

- ▶ A poorly-cohesive method/function could also be doing several things *in sequence*.
  - ▶ It doesn't always have to involve a control flag and an if statement.
  - ▶ It could simply do *all* of the tasks.
- ▶ This is still bad, and the method/function should still be split up as before.

```
public static String[] formatTimeDate(int one,  Java
                                     int two, int three) {
    String[] s = new String[2];
    s[0] = one + "/" + two + "/" + three;
    s[1] = one + ":" + two + ":" + three;
    return s; // Return an array containing both results
}
```

## Sequential Tasks

- ▶ A poorly-cohesive method/function could also be doing several things *in sequence*.
  - ▶ It doesn't always have to involve a control flag and an if statement.
  - ▶ It could simply do *all* of the tasks.
- ▶ This is still bad, and the method/function should still be split up as before.

```
def formatTimeDate(one, two, three):  
    s0 = str(one) + "/" + str(two) + "/" + str(three)  
    s1 = str(one) + ":" + str(two) + ":" + str(three)  
    return (s0, s1) # Return a tuple containing both results
```



## Relatedness of Tasks

- ▶ Even among methods/functions that perform multiple tasks, there are varying levels of cohesion.
- ▶ The degree of cohesion depends on how related the tasks are to each other:
  - ▶ Completely unrelated – extremely low (essentially zero) cohesion.
  - ▶ Superficially related by name or some ad hoc category.
  - ▶ Related by time – the tasks must be performed at about the same time, perhaps in a particular order.
  - ▶ Related by data – the tasks all use the same data, perhaps data produced by each other.

## Different Data

- If distinct parts of a method/function use different data, it probably has poor cohesion.

```
public static void checkAgeAndPostcode(int age, int postcode)
{
    if(0 <= age && age <= 130) {
        System.out.println("Valid age");
    }
    else {
        System.out.println("Invalid age");
    }
    if(1000 <= postcode && postcode < 10000) {
        System.out.println("Valid postcode");
    }
    else {
        System.out.println("Invalid postcode");
    }
}
```



## Different Data

- ▶ If distinct parts of a method/function use different data, it probably has poor cohesion.

```
def checkAgeAndPostcode(age, postcode):  
    if 0 <= age <= 130:  
        print("Valid age")  
    else:  
        print("Invalid age")  
  
    if 1000 <= postcode < 10000:  
        print("Valid postcode")  
    else:  
        print("Invalid postcode")
```





## Different Data – Discussion

- ▶ `checkAgeAndPostcode()` has two parts that work with different data.
- ▶ Therefore, it is clearly performing two different tasks – low cohesion.
- ▶ Why is this bad?
  - ▶ What if you want to check *only* the age, or *only* the postcode?
  - ▶ You can't do it with this method/function.
  - ▶ If there were *two separate* methods/functions, you could.

## Refactoring to Improve Cohesion

If a module performs several unrelated tasks, break it up:

```
public static void checkAge(int age) {  
    if(0 <= age && age <= 130) {  
        System.out.println("Valid age");  
    } else {  
        System.out.println("Invalid age");  
    }  
}
```



```
public static void checkPostcode(int postcode) {  
    if(1000 <= postcode && postcode < 10000) {  
        System.out.println("Valid postcode");  
    } else {  
        System.out.println("Invalid postcode");  
    }  
}
```

## Refactoring to Improve Cohesion

If a module performs several unrelated tasks, break it up:

```
def checkAge(age):  
    if 0 <= age <= 130:  
        print("Valid age")  
    else:  
        print("Invalid age")  
  
def checkPostcode(postcode):  
    if 1000 <= postcode < 10000:  
        print("Valid postcode")  
    else:  
        print("Invalid postcode")
```



## Refactoring to Improve Cohesion (continued)

Find where you called the original method/function:

```
checkAgeAndPostcode(someAge, somePostcode)
```

And break up the call(s) as well:

```
checkAge(someAge)  
checkPostcode(somePostcode)
```

- ▶ This won't affect the functionality.
- ▶ It *will* improve cohesion (and hence flexibility, maintainability, etc.).

## Redundancy (or Repetition or Duplication)

- ▶ Good software design seeks to avoid redundancy, repetition, duplication, repetition, repetition and redundancy.
- ▶ Code is redundant if it performs a task that is already performed by another piece of code.
- ▶ Redundancy is good in hardware:
  - ▶ Physical things wear out over time and become faulty.
  - ▶ Duplication of physical parts can improve reliability.
  - ▶ Unlikely that they will all fail simultaneously.
- ▶ Redundancy is (usually) bad in software:
  - ▶ Software *does not* wear out over time.
  - ▶ Duplicate software systems are guaranteed to fail simultaneously (under the same conditions).
  - ▶ Redundancy increases complexity without any benefit.
- ▶ The opposite of redundancy is *reuse*.

## Benefits of Reuse

- ▶ Redundancy increases the amount of code unnecessarily.
- ▶ All else being equal, a small system is better than a large one.
  - ▶ Easier to test – fewer test cases.
  - ▶ Easier to inspect – less material to review.
  - ▶ Less fault-prone – less opportunity for making mistakes.
  - ▶ Easier to maintain – less to understand.
- ▶ Some systems *must* be large, because their requirements are large, but they should not be any larger than necessary.
- ▶ Software engineers don't get paid per line of code.
  - ▶ (If they do, the project is doomed to be a catastrophe of useless, incomprehensible code.)
- ▶ As a software engineer, some of your best work may be *removing* code, rather than adding more of it!

## But But But...

- ▶ You may be thinking:
  - ▶ “Those test cases I’ve been writing seem awfully repetitive.”
- ▶ Yes, they do!
- ▶ We briefly mentioned how to use loops and arrays to avoid that sort of repetition.
- ▶ However, some repetition is indeed unavoidable, due to:
  - ▶ The nature of the language.
  - ▶ The development environment.
  - ▶ The standards set by your organisation.
- ▶ Zero repetition is the “unobtainium” of software design.


## Refactoring Redundancy – Reusing Modules

- ▶ We always try our best to minimise redundancy.
- ▶ If modules A and B perform exactly the same task:
  - ▶ One should be deleted; e.g. B.
  - ▶ Any other modules using B should instead use A.
- ▶ If module A is a superset of module B:
  - ▶ The duplication should be deleted from A.
  - ▶ Module A should instead *use* module B (rather than duplicate it).
- ▶ If modules A and B (and maybe even C, D, etc.) perform overlapping tasks:
  - ▶ Identify the overlapping code.
  - ▶ Delete it from both A and B (and C, D, etc. if applicable).
  - ▶ Create a new module Z, containing the overlapping code.
  - ▶ Have the other modules use module Z.




# Supersets

```
public static boolean checkValid(int x) {  
    return 0 <= x && x < 100;  
}  
public static void printIfValid(int number) {  
    if(number >= 0 && number < 100)  
        System.out.println(number);  
}  
// Notice that the two highlighted  
// sections are equivalent.
```




Refactor printIfValid() to call checkValid():

```
public static void printIfValid(int number) {  
    if(number >= 0 && number < 100 checkValid(number))  
        System.out.println(number);  
}  
}
```




# Supersets

```
def checkValid(x):  
    return 0 <= x < 100  
  
def printIfValid(number):  
    if number >= 0 and number < 100:  
        print(number)      # Notice that the two highlighted  
                           # sections are equivalent.
```



Refactor printIfValid() to call checkValid():

```
def printIfValid(number):  
    if number >= 0 and number < 100 checkValid(number):  
        print(number)
```



## Common Tasks

```
public static void printSpecial(double x, double y) {  
    if(3 * x * x * y * (x - y) > 0.0) {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}  
  
public static double getNM() {  
    double n = 0.0;  
    double m = 0.0;  
    while(n <= 0.0 || m <= 0.0) {  
        n = ...; // Input n value  
        m = ...; // Input m value  
    }  
    return (n - m) * m * n * 3 * n;  
}
```



- Remove the common code, and put it in a new method.

## Common Tasks

```
def printSpecial(x, y):  
    if 3 * x * x * y * (x - y) > 0.0:  
        print(x, y)  
  
def getNM():  
    n = 0.0  
    m = 0.0  
    while n <= 0.0 or m <= 0.0:  
        n = float(input())  
        m = float(input())  
    return (n - m) * m * n * 3 * n
```



- Remove the common code, and put it in a new function.

## Common Tasks Refactored

```
public static double calcXY(double x, double y) {  
    return 3 * x * x * y * (x - y);  
} // new method
```



```
public static void printSpecial(double x, double y) {  
    if(calcXY(x, y)) {  
        System.out.println("(" + x + "," + y + ")");  
    }  
}
```

```
public static double getNM() {  
    double n = 0.0;  
    double m = 0.0;  
    while(n <= 0.0 || m <= 0.0) {...}  
    return calcXY(n, m);  
}
```

## Common Tasks Refactored



```
def calcXY(x, y): # new function
    return 3 * x * x * y * (x - y)
```

```
def printSpecial(x, y):
    if calcXY(x, y) > 0:
        print(x, y)
```

```
def getNM():
    n = 0.0
    m = 0.0
    while n <= 0.0 or m <= 0.0:
        ...
    return calcXY(n, m)
```

## Reuse and Coupling

- ▶ Reuse (a good thing) actually increases coupling (a bad thing).
  - ▶ A slight paradox, or rather a balancing act.
    - ▶ Sensible reuse does not cause *undue* coupling.
    - ▶ Sensible coupling does not cause *undue* redundancy.
  - ▶ If you're *not* sensible, you might:
    - ▶ See duplication where there isn't any.
    - ▶ Try to “reuse” things that are not applicable.
- These will increase coupling unnecessarily (and possibly also reduce cohesion).

Design  
oooo

Maintenance  
oooo

Coupling  
oooooooooooooooo

Cohesion  
oooooooooo

Redundancy  
oooooooo●

That's all for now!