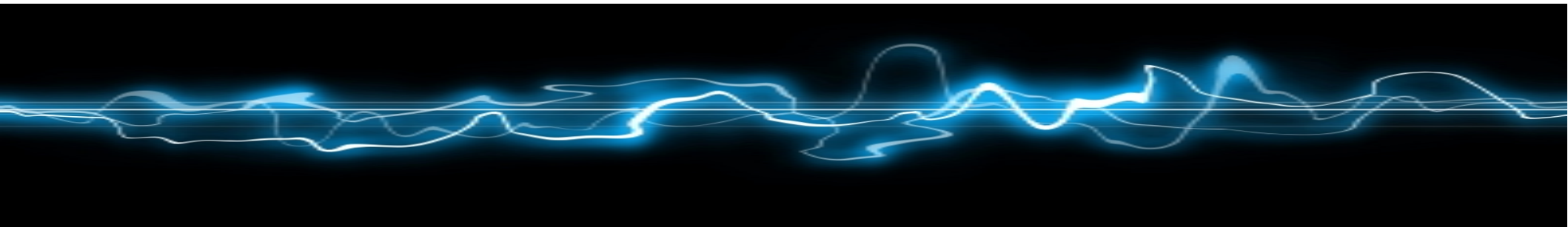


## **Core principals of Software design.**

**Goals: Looking at cohesion and coupling in software design.  
Static variables and methods. Accessors and mutators.**



# This term

Every Tuesday 9-10 am is tutorial time, ask one-to-one questions, plus catch-up on arrays, classes etc. Come early (9 am) if you want a tutorial.

Week 4, 31<sup>st</sup> 10-1: software design, Sound workshop, midi, synthesis, sampling, linking to Pure Data, Max etc  
Week 5, 7<sup>th</sup> February 10-1: **5 minute interim presentations** on how your projects are going.  
Databases and Processing if there is time.

## Week 6 READING WEEK

Week 7, 21<sup>st</sup> Feb: Android development with Processing, bring Android phones, usb leads and tablets  
**if Android 2.1 +**

Week 8, 28<sup>th</sup> Feb 10-1: **3D in Processing**

Week 9, 6<sup>th</sup> March 10-1: Java documentaiton. 5 minute **interim presentations** on how your projects are going, trouble shoot.

Week 10, 13<sup>th</sup> March 10-1: get on with projects, trouble shoot.

Week 11 20<sup>th</sup> 10-1: Hand in work online and present

# Class/static variables

**The key thing about static variables or methods is that they are independent of instances of objects.**

It is a variable which **belongs to the class** and **not** to an **object** (instance)

Static variables are **initialized only once**, at the start of the execution.

Static variables will be initialized first, before the initialization of any instance variables

There is a **single copy** to be shared by all instances of the class

A static variable can be **accessed directly** by the **class name** and doesn't need any object

# Static methods

## *static method*

- ✓ It is a method which **belongs to the class** and **not** to the **object**(instance)
- ✓ A static method **can access only static data**. It can not access non-static data (instance variables)
- ✓ A static method **can call only** other **static methods** and can not call a non-static method from it.
- ✓ A static method can be **accessed directly** by the **class name** and doesn't need any object
- ✓ Syntax : **<class-name>.<method-name>**
- ✓ A static method cannot refer to "this" or "super" keywords in anyway

```
Circle[] circ;  
Circle aCirc;
```

```
static int numberOfCircles= 0;  
/*only one of these variables in the whole program  
its independent of instance variables */
```

```
void setup() {
```

```
size(400, 400);  
circ = new Circle[5];
```

```
for (int i = 0; i < circ.length; i ++ ) {  
  circ[i] = new Circle(random(width), random(height));  
}
```

```
void draw() {
```

```
for (int i = 0; i < circ.length; i ++ ) {  
  circ[i].drawCircle();  
}
```

```
if (mousePressed) {  
  aCirc = new Circle(mouseX, mouseY);  
}
```

```
if (aCirc!=null) {  
  aCirc.drawCircle();  
}
```

```
println(numberOfCircles); //static variable keeps track of Circle instances
```

```
class Circle {
```

```
float a, b;
```

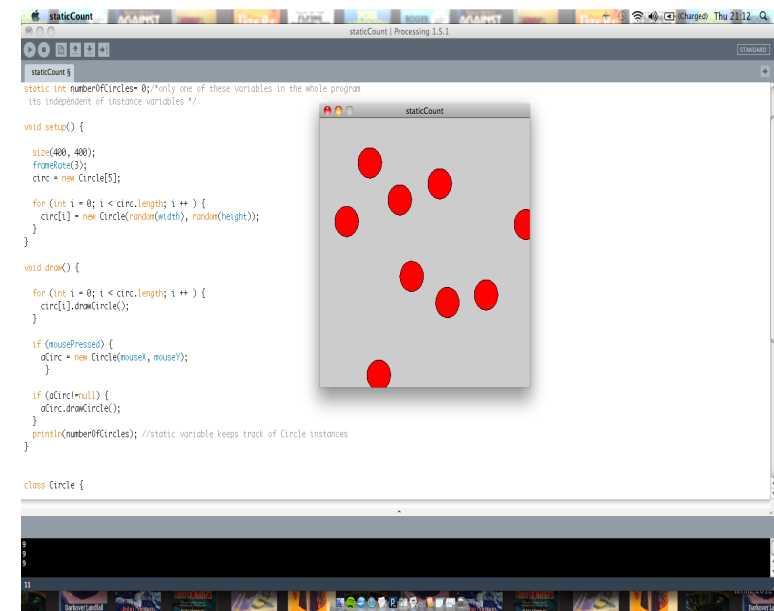
```
Circle(float tempA, float tempB) {  
  a = tempA;  
  b = tempB;  
  numberOfCircles++;  
  /*everytime the keyword 'new' is used to instantiate a  
  Circle object this number goes up*/  
}
```

```
void drawCircle() {  
  fill(255, 0, 0);  
  ellipse(a, b, 45, 45);  
}
```

```
} //end Circle class
```

A static variable (also called a 'class' variable) is independent of object instances, there is only one copy of it in the program, here I use a static variable to keep track of instances of the Circle class

Everytime the keyword 'new' is used to instantiate a Circle object the static int **numberOfCircles** goes up by one.



```
Circle[] circ;
Circle aCirc;

static int numberOfCircles= 0; /*only one of these variables in the whole program
its independent of instance variables */

void setup() {
  size(400, 400);
  frameRate(10);
  circ = new Circle[5];

  for (int i = 0; i < circ.length; i ++ ) {
    circ[i] = new Circle(random(width), random(height));
  }
}

void draw() {
  for (int i = 0; i < circ.length; i ++ ) {
    circ[i].drawCircle();
  }

  if (mousePressed) {
    aCirc = new Circle(mouseX, mouseY);
  }

  if (aCirc!=null) {
    aCirc.drawCircle();
  }
  staticClass.count();//address through name of class
  //no need for an instance of staticClass as it is static
}

class staticClass {

  static void count() {
    println(numberOfCircles); //static variable keeps track of Circle instances
  }
}

class Circle {

  float a, b;

  Circle(float tempA, float tempB) {
    a = tempA;
    b = tempB;
    numberOfCircles++;
    /*everytime the keyword 'new' is used to instantiate a
    Circle object this number goes up*/
  }

  void drawCircle() {
    fill(255, 0, 0);
    ellipse(a, b, 45, 45);
  }
}
//end Circle class
```

## Class staticCountMethod

Here I've added a static class and a static method that prints the number of Circle instances currently in the program. A static class is not designed to be instantiated but addressed directly through the name of the class: **StaticClass.count();**

**Note Java and Processing behave in slightly different ways with static fields and classes. You wouldn't normally declare a class as static in Java, but I had to In Processing to make this example work.**

## Java example, similarly uses a static variable to count instances of the Student class

```
Compile Undo Cut Copy Paste Find... Close Source Code
```

```
class Student {
int a; //initialized to zero
static int b; //initialized to zero only when class is loaded not for each object created.

Student(){
//Constructor incrementing static variable b
b++;
}

public void showData(){
System.out.println("Value of a = "+a);
System.out.println("Value of b = "+b);
}

//public static void increment(){
//a++;
//}
}

class Demo{
public static void main(String args[]){
Student s1 = new Student();
s1.showData();
Student s2 = new Student();
s2.showData();
//Student.b++;
//s1.showData();
}
}
```

BlueJ: Terminal Window - examples

```
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```

'staticDemo' in examples in folder

# **Core Software Design Principals**

# Cohesion

**Cohesion describes how well a unit of code maps to a logical task**

**Good class design exhibits a high degree of cohesion.**

**Ideally one unit of code should be responsible for one cohesive task.**

**A method should implement one logical operation, and a class should represent one type of entity (a mountain bike, a Cat, a sound loader, an ellipse etc)**

# Cohesion and coupling

In computer science, **coupling** or dependency is the degree to which each program module relies on each one of the other modules/methods/classes.

**Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:**

**A change in one module usually forces a ripple effect of changes in other modules.**

**Assembly of modules might require more effort and/or time due to the increased inter-module dependency.**

**A particular module might be harder to reuse and/or test because dependent modules must be included.**

**The degree of coupling will determine how hard it is to make changes to an application.**

**If a class is tightly coupled it makes it much harder to make changes to that class without effecting other classes.**

# Software Design Guidelines

Common advice to beginners about writing good object-oriented programs is, “Don’t put too much into a single method,” or “Don’t put everything into one class.” Both suggestions have merit, but frequently lead to the counter questions, “How long should a method be?” or “How long should a class be?”

These questions can be answered in terms of **cohesion and coupling**

In brief:

A method is too long if it does more than one logical task.

A class is too complex if it represents more than one logical entity.

These answers do not give clear-cut rules that specify what exactly to do.

Terms such as *one logical task* are still open to interpretation, and different programmers will decide differently in many situations.

These are guidelines (not cast-in-stone rules).

Keeping these guidelines in mind, though, will significantly improve your class design and enable you to tackle more complex problems and write better and more interesting programs.

# What are design patterns?

Software architects and developers leverage a core set of design principles that have been established by professionals over the course of many years. Many of these principles are foundational to the most well-known software design patterns. While a software system may have its own set of unique principles to guide its design, it is believed that all systems can benefit from the use of this core set.

## Separate code that varies from code that stays the same

If you have some aspect of your code that tends to change frequently, consider separating that code from the code that does not change. In other words, you should encapsulate the parts that vary so that you can later alter or extend those parts without touching the parts that do not vary.

This concept may be simple, but it is extremely important; it forms the basis for almost every software design pattern.

## Program to an interface, not an implementation

When you program to concrete implementations of classes, you get locked into those concrete implementations. If you program to an interface instead, your only reliance is on the interface and not the implementation. So, you're not locked in; you can swap out the implementation of some aspect of your program more easily.

**The word *interface* has at least three different meanings in the context of programming**

# Loose coupling

## Strive for loose coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

Loosely coupled designs allow us to build systems that are more flexible because they minimize the interdependency between objects.

## Favour composition over inheritance

Consider the HAS-A relationship: a Person has a WalkBehavior and a TalkBehavior and the person delegates talking and walking to these behaviors.

When you put two classes like this together, you are using **composition**. Instead of *inheriting* her behavior, the Person gets her behavior by being *composed* with the right behavior objects.

Using composition in lieu of inheritance gives you greater flexibility. It lets you encapsulate a family of algorithms into their own set of classes and it also lets you change behavior at runtime as long as the object you've composed implements the correct behavior interface.

Composition is used in many design patterns. The strategy pattern in particular, promotes composition over inheritance.

In computing and systems design a **loosely coupled system** is one where each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. The notion was introduced into organizational studies by Karl Weick. Sub-areas include the coupling of classes, interfaces, data, and services.

# Classes should be open for extension, but closed for modification (open-closed principle)

Classes should be easily extended to incorporate new behavior without modifying existing code. When we accomplish this, our designs are resilient to change and flexible enough to take on new functionality and meet changing requirements.

# Depend upon abstractions, not concretes

This is also known more formally as the *Dependency Inversion Principle*. It is similar to the concept of programming to an interface rather than an implementation, but it makes an even stronger statement about abstraction. It suggests that high-level components should not depend on low-level components; they should both depend on abstractions. One example can be visualized by imagining one abstract class inserted between a parent object that contains many child objects. Instead of making the parent refer to the child objects directly, it refers to their abstract base class.

# **Principle of Least Knowledge - interact only with your immediate friends**

For any object in a system, you need to be careful of the number of classes it interacts with and how it comes to interact with those classes.

This principle prevents us from creating systems that have a large number of classes coupled together where changes in one part of the system cascade to other parts. A system with too many dependencies is expensive to maintain and difficult for others to understand.

# **The Hollywood Principle - Don't call us, we'll call you**

Low-level components can participate in a computation, but the high-level components should control when and how. A low-level component should never call a high-level component directly.

# A class should have only one reason to change

Assign each responsibility to one class, and only one class.

Modifying code provides opportunities for new problems to arise. If a class has many ways to change, the probability of affecting multiple aspects of a system are increases when the class is changed.

This principle might also be called *Separation of Responsibility*, but we think our chosen heading expresses more. Separating responsibility sounds deceptively simple, but it's can actually be quite hard to do. The human brain is design to classify, organize, and group concepts, so doing the exact opposite can be a bit counter-intuitive.

# Design to avoid Rigidity, Fragility, and Immobility

In his paper, "Dependency of Inversion Principal", Robert C. Martin wrote the following which I think makes a perfect conclusion to this article.

*But there is one set of criteria that I think all engineers will agree with. A piece of software that fulfills its requirements and yet exhibits any or all of the following traits has a bad design.*

*It is hard to change because every change affects too many other parts of the system. (Rigidity)*

*When you make a change, unexpected parts of the system break. (Fragility)*

*It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)*

*Moreover, it would be difficult to demonstrate that a piece of software that exhibits none of those traits, i.e. it is flexible, robust, and reusable, and that also fulfills all its requirements, has a bad design. Thus, we can use these three traits as a way to unambiguously decide if a design is 'good' or 'bad'.*

Keeping all of these principles in mind, as we design, we can create systems that are more easily understood and modified. We can create systems that are less rigid, less fragile, and that are assembled by reusable parts that can be leveraged again and again.

# Responsibility-driven-design

Object oriented design is the art of assigning the right responsibilities to the right objects and creating a clear structure with loose coupling and high cohesion. Test driven development (TDD) is a design practice that helps you to achieve this to some extent. TDD drives you towards loosely coupled objects, because too many dependencies will hinder the short test-code-refactor cycles typical for TDD.

Responsibility driven design is an approach that helps you shift focus from object state to interactions and responsibilities.

**Responsibility-driven design expresses the idea that each class should be responsible for handling its own data. Often, when we need to add some new functionality to an application, we need to ask ourselves in which class we should add a method to implement this new function.**

**Which class should be responsible for the task? The answer is that the class that is responsible for storing some data should also be responsible for manipulating it.**

**How well responsibility-driven design is used influences the degree of coupling, and therefore, again, the ease with which an application can be modified or extended.**

**Another aspect of the de-coupling and responsibility principles is that of localizing change.**

**We aim to create a class design that makes later changes easy by localizing the effects of a change. Ideally, only a single class needs to be changed to make a modification.**

**Sometimes several classes need change, but then we aim at this being as few classes as possible. In addition, the changes needed in other classes should be obvious, easy to detect, and easy to carry out.**

**To a large extent, we can achieve this by following good design rules such as using responsibility-driven design and aiming for loose coupling and high cohesion.**

## **Refactoring**

**Refactoring is the activity of restructuring existing classes and methods to adapt them to changed functionality and requirements.**

**Often, in the lifetime of an application, functionality is gradually added. One common effect is that, as a side effect of this, methods and classes slowly grow in length.**

**It is tempting for a maintenance programmer to add some extra code to existing classes or methods. Doing this for some time, however, decreases the degree of cohesion.**

**When more and more code is added to a method or a class, it is likely that at some stage it will represent more than one clearly defined task or entity.**

**Refactoring is the re-thinking and re-designing of class and method structures. Most commonly the effect is that classes are split into two, or that methods are divided into two or more methods.**

**Refactoring can also include the joining of classes or methods into one, but that case is less common.**

We need to understand **encapsulation** to make more sense of some of the design principals we've just covered.

We'll look at getters and setters (also called *Accessors* and *Mutators*)

In short **encapsulation** is a way of keeping access to variables to a minim, it reduces tight coupling.

This is considered a good (and safe) design practice.

**Getters and Setters  
also called Accessors and Mutators**

In Java *getters* and *setters*, also known as accessors and mutators, are central tools in the implementation of ‘encapsulation’ or data hiding – keeping access to variables as limited as practically possible

Processing treats classes on other tabs as inner classes, so you will not see getters and setters used so often, but there are times when you still may use this method

In Java **data hiding**, or **encapsulation**, is a central concept. Encapsulation keeps your variables within a minimum degree of visibility. The idea is to protect them from having incorrect or accidental values assigned to them. It also makes debugging easier, as you can more clearly trace where and when the values were assigned.

It is considered good programming style to keep public fields to a minimum, to make your fields(see note\*) private and grant access to those fields selectively by creating **getter** and **setter** methods. You could also, in suitable circumstances omit a setter method so a value cant be changed, this would make it a read only property (there are other ways, such as creating constant variables with the word 'final', eg: **final int WEEKDAYS = 5;** this cant be changed once its been initialized) Setter methods can also be used to check that a value is in the correct range for your purposes (eg for a survey or game or whatever)

**\*A Field is a variable that is defined in the body of a class, but outside of any of that class's methods, if they do not have the word static in front of them they can also be called *instance variables*, The modifier ' private' means that variable is not visible outside of the class, 'public' means it is, 'protected' means it is visible to sub classes (unlike private) but not to outside classes. An instance variable exists to be used when you create an instance of a class (or object)**

***static* means the variable or method is associated with the class not with an instance of a class, you don't have to create an object to use a static method or address a static variable (but you *can* access a static method or field from an object instance if you want to.....)**

In this way a setter method is validating the data, for example to check that an int is not less than zero or greater than 10, if you had an online survey or scoring mechanism of some sort you could build code like this into the setter:

```
public void setScore(int s)
```

```
{
```

```
if (s <0)
```

```
    score = 0;
```

```
else if (s >10)
```

```
    score = 10;
```

```
else
```

```
score = s;
```

```
}
```

```
//full code coming up soon
```

We have used lots of built in getter and setter methods in  
In Java and a couple in Processing, for example:

```
g. setColor(Color.red);  
setSize(200, 200);  
Container contentArea = getContentPane();  
contentArea.setBackground(Color.cyan);  
slider.getValue();
```

In Processing **set** and **get** methods are used for reading and for writing pixels:

```
get(x, y);  
set(x, y, color);
```

Here's a Java example (also in examples folder):

```
class mutantMonster
{
private String color;
private String body;

//constructor for mutantMonster class, these are default values:
public mutantMonster()
{
    color = ("orange and pink");
    body = ("slimey and cold");

}

public void setMonster(String col, String bod)    //setter or 'mutator' method
{
//this could also be broken down into two methods, one for each variable
    color = col;
    body = bod;
}

public String describeMonster()
//getter or 'accessor' method
{
return("Monster colour is "+ color +", the monster's body is "+ body);
//we could break this down into two methods
}
}
//end of mutantMonster class
```

```
public class monsterData
{
public static void main(String[] args)
    {
mutantMonster monsto = new mutantMonster(); //create an instance of mutantMonster class

//use the setter method of mutantMonster to change values:
monsto.setMonster("Green","Scaley with tentacles");
//comment out above to get default values from mutantMonster constructor

//use accessor or getter method of mutantMonster class to access those values:

System.out.println(monsto.describeMonster());

//try to execute these statements - what happens?
//monsto.color = "purple";
//mutantMonster.color = "green";
//System.out.println(monsto.color);
//System.out.println(mutantMonster.color);
    }
}
```

By declaring the object (or instance) variables 'color' and 'body' to be **private** in the mutantMonster class we protect them from being **directly** changed by **external** program code, we have provided a setter or 'mutator' method to allow the variables to be **manipulated** (mutated!), and a getter (or accessor) method to get access to those variables.

That is **encapsulation**.

We don't have to call our own methods '**getSomething()**' or '**setSomething()**' but it helps to call them names that let us and others understand exactly what they do, **describeMonster()** In our example seems more understandable then **getMonster()** but **getMonster()** would be a more conventional naming practice

These names are clear but a bit too awkward would you agree?

**getMonster\_Data()**

**getMonsterDetails()**

Its up to you to think of good names in relation to **your own**

Accessor/mutator methods, the words **get** and **set** are just conventions

Run the previous code and do the suggested experiments - and any others that occur to you...

Full version of code that uses a setter method to validate data:

```
class Score
{
private int score;

public void setScore(int s) //setter method with validating algorithm:
{
if (s <0)           //if s is less than zero correct value to 0
    score = 0;
else if (s >10)     //if s is greater than 10 correct value to 10
    score = 10;
else
score = s;    //otherwise accept value
}

public int getScore()
//getter or 'accessor' method:
{
return(score);
}
}
```

```
public class scoreBoard
{

public static void main(String[] args)
{
Score score1 = new Score();
score1.setScore(-3);
System.out.println(score1.getScore());

}
}
```

# Do you want to know more?

Following are some of the best resources about software design patterns and principles

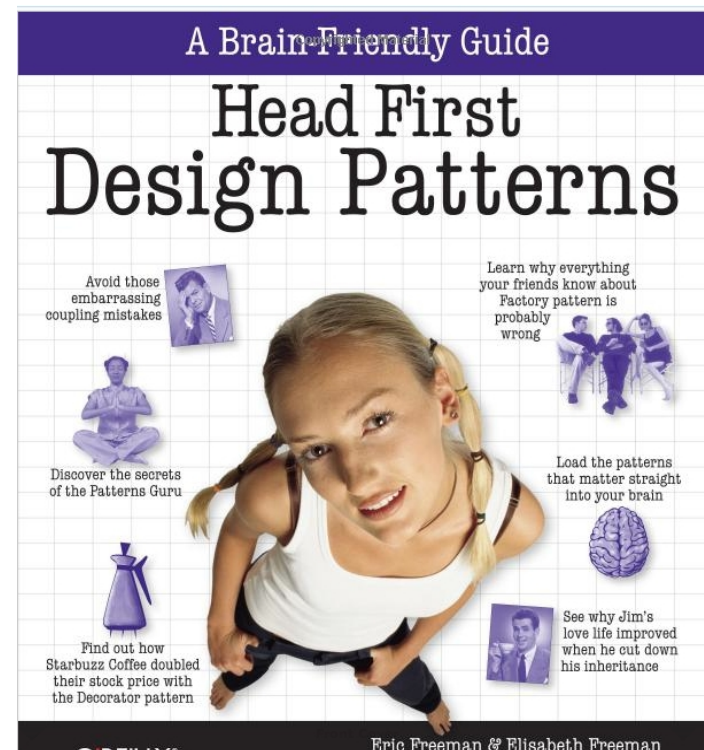
Freeman, Eric & Elisabeth. [Head First Design Patterns](#). O'Reilly Media, Inc., 2004.

Design Patterns - at Wikipedia

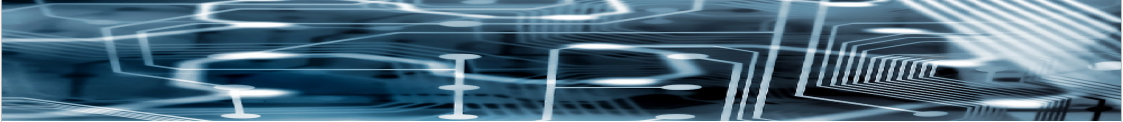
Patterns Library - at Hillside.net

Patterns - at the ServerSide.com

[J2EE Patterns Catalog](#)

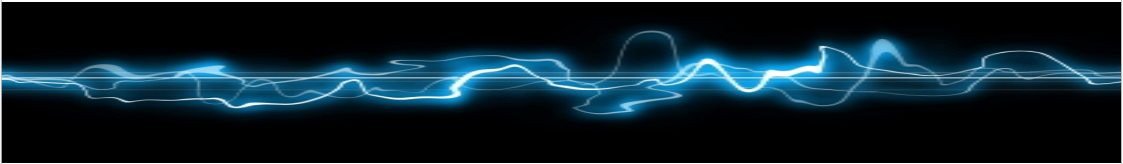


<https://wiki.base22.com/display/btg/Core+Software+Design+Principles>



## **Core principals of Software design.**

**Goals: Looking at cohesion and coupling in software design.  
Static variables and methods. Accessors and mutators.**





## Class/static variables

**The key thing about static variables or methods is that they are independent of instances of objects.**

It is a variable which **belongs to the class** and **not** to an **object** (instance)

Static variables are **initialized only once**, at the start of the execution.

Static variables will be initialized first, before the initialization of any instance variables

There is a **single copy** to be shared by all instances of the class

A static variable can be **accessed directly** by the **class name** and doesn't need any object

# Static methods

## ***static method***

- ✓ It is a method which **belongs to the class** and **not** to the **object**(instance)
- ✓ A static method **can access only static data**. It can not access non-static data (instance variables)
- ✓ A static method **can call only** other **static methods** and can not call a non-static method from it.
- ✓ A static method can be **accessed directly** by the **class name** and doesn't need any object
- ✓ Syntax : **<class-name>.<method-name>**
- ✓ A static method cannot refer to "this" or "super" keywords in anyway

```

Circle[] circ;
Circle aCirc;

static int numberOfCircles= 0;
/*only one of these variables in the whole program
its independent of instance variables */

void setup() {
  size(400, 400);
  circ = new Circle[5];
  for (int i = 0; i < circ.length; i ++ ) {
    circ[i] = new Circle(random(width), random(height));
  }
}

void draw() {
  for (int i = 0; i < circ.length; i ++ ) {
    circ[i].drawCircle();
  }

  if (mousePressed) {
    aCirc = new Circle(mouseX, mouseY);
  }

  if (aCirc!=null) {
    aCirc.drawCircle();
  }
  println(numberOfCircles); //static variable keeps track of Circle instances
}

class Circle {

  float a, b;

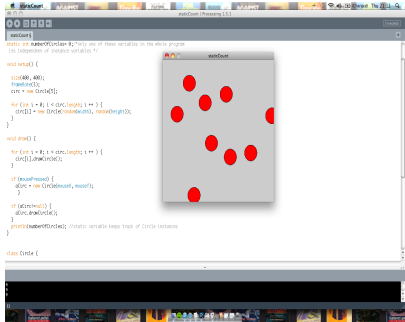
  Circle(float tempA, float tempB) {
    a = tempA;
    b = tempB;
    numberOfCircles++;
    /*everytime the keyword 'new' is used to instantiate a
    Circle object this number goes up*/
  }

  void drawCircle() {
    fill(255, 0, 0);
    ellipse(a, b, 45, 45);
  }
}
//end Circle class

```

A static variable (also called a 'class' variable) is independent of object instances, there is only one copy of it in the program, here I use a static variable to keep track of instances of the Circle class

Everytime the keyword 'new' is used to instantiate a Circle object the static int numberOfCircles goes up by one.



```

Circle[] circ;
Circle aCirc;

static int numberOfCircles= 0; /*only one of these variables in the whole program
its independent of instance variables */

void setup() {
  size(400, 400);
  frameRate(10);
  circ = new Circle[5];

  for (int i = 0; i < circ.length; i++) {
    circ[i] = new Circle(random(width), random(height));
  }
}

void draw() {
  for (int i = 0; i < circ.length; i++) {
    circ[i].drawCircle();
  }

  if (mousePressed) {
    aCirc = new Circle(mouseX, mouseY);
  }

  if (aCirc!=null) {
    aCirc.drawCircle();
  }
  staticClass.count();//address through name of class
  //no need for an instance of staticClass as it is static
}

class staticClass {
  static void count() {
    println(numberOfCircles); //static variable keeps track of Circle instances
  }
}

class Circle {

  float a, b;

  Circle(float tempA, float tempB) {
    a = tempA;
    b = tempB;
    numberOfCircles++;
    /*everytime the keyword 'new' is used to instantiate a
    Circle object this number goes up*/
  }

  void drawCircle() {
    fill(255, 0, 0);
    ellipse(a, b, 45, 45);
  }
} //end Circle class

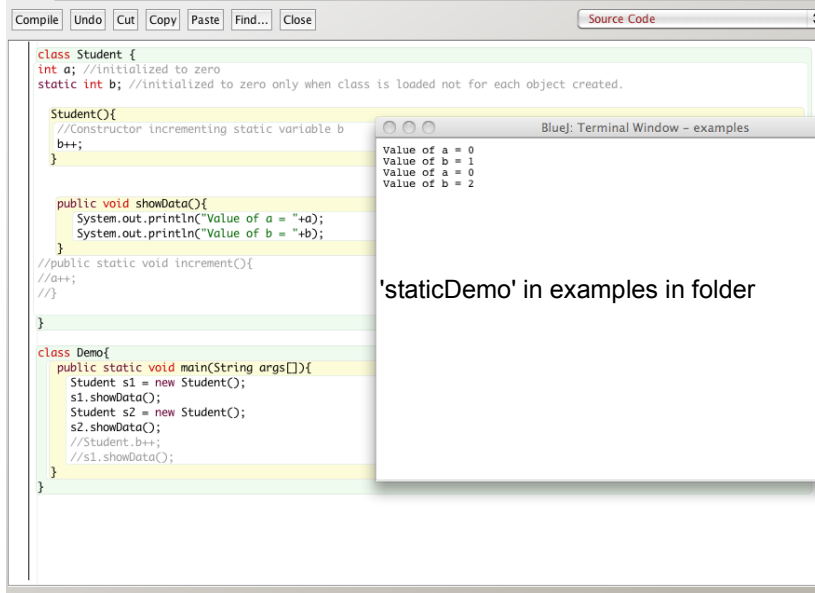
```

### Class staticCountMethod

Here I've added a static class and a static method that prints the number of Circle instances currently in the program. A static class is not designed to be instantiated but addressed through the name of the class: **StaticClass.count();**

**Note Java and Processing behave in slightly different ways with static fields and classes. You wouldn't normally declare a class as static in Java, but I had to In Processing to make this example work.**

Java example, similarly uses a static variable to count instances of the Student class



```
Compile Undo Cut Copy Paste Find... Close Source Code
```

```
class Student {
    int a; //initialized to zero
    static int b; //initialized to zero only when class is loaded not for each object created.

    Student(){
        //constructor incrementing static variable b
        b++;
    }

    public void showData(){
        System.out.println("Value of a = "+a);
        System.out.println("Value of b = "+b);
    }

    //public static void increment(){
    //a++;
    //}
}

class Demo{
    public static void main(String args[]){
        Student s1 = new Student();
        s1.showData();
        Student s2 = new Student();
        s2.showData();
        //Student.b++;
        //s1.showData();
    }
}
```

Bluej: Terminal Window - examples

```
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```

'staticDemo' in examples in folder

# **Core Software Design Principals**

## **Cohesion**

**Cohesion describes how well a unit of code maps to a logical task**

**Good class design exhibits a high degree of cohesion.**

**Ideally one unit of code should be responsible for one cohesive task.**

**A method should implement one logical operation, and a class should represent one type of entity (a mountain bike, a Cat, a sound loader, an ellipse etc)**

## Cohesion and coupling

In computer science, **coupling** or dependency is the degree to which each program module relies on each one of the other modules/methods/classes.

Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:

A change in one module usually forces a ripple effect of changes in other modules.

Assembly of modules might require more effort and/or time due to the increased inter-module dependency.

A particular module might be harder to reuse and/or test because dependent modules must be included.

The degree of **coupling** will determine how hard it is to make changes to an application.

If a class is tightly coupled it makes it much harder to make changes to that class without effecting other classes.

## Software Design Guidelines

Common advice to beginners about writing good object-oriented programs is, “Don’t put too much into a single method,” or “Don’t put everything into one class.” Both suggestions have merit, but frequently lead to the counter questions, “How long should a method be?” or “How long should a class be?”

These questions can be answered in terms of **cohesion and coupling**

In brief:

**A method is too long if it does more than one logical task.**

**A class is too complex if it represents more than one logical entity.**

These answers do not give clear-cut rules that specify what exactly to do.

Terms such as *one logical task* are still open to interpretation, and different programmers will decide differently in many situations.

These are guidelines (not cast-in-stone rules).

Keeping these guidelines in mind, though, will significantly improve your class design and enable you to tackle more complex problems and write better and more interesting programs.

# What are design patterns?

Software architects and developers leverage a core set of design principles that have been established by professionals over the course of many years. Many of these principles are foundational to the most well-known software design patterns. While a software system may have its own set of unique principles to guide its design, it is believed that all systems can benefit from the use of this core set.

## Separate code that varies from code that stays the same

If you have some aspect of your code that tends to change frequently, consider separating that code from the code that does not change. In other words, you should encapsulate the parts that vary so that you can later alter or extend those parts without touching the parts that do not vary.

This concept may be simple, but it is extremely important; it forms the basis for almost every software design pattern.

## Program to an interface, not an implementation

When you program to concrete implementations of classes, you get locked into those concrete implementations. If you program to an interface instead, your only reliance is on the interface and not the implementation. So, you're not locked in; you can swap out the implementation of some aspect of your program more easily.

**The word *interface* has at least three different meanings in the context of programming**

# Loose coupling

## Strive for loose coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

Loosely coupled designs allow us to build systems that are more flexible because they minimize the interdependency between objects.

## Favour composition over inheritance

Consider the HAS-A relationship: a Person has a WalkBehavior and a TalkBehavior and the person delegates talking and walking to these behaviors.

When you put two classes like this together, you are using **composition**. Instead of *inheriting* her behavior, the Person gets her behavior by being *composed* with the right behavior objects.

Using composition in lieu of inheritance gives you greater flexibility. It lets you encapsulate a family of algorithms into their own set of classes and it also lets you change behavior at runtime as long as the object you've composed implements the correct behavior interface.

Composition is used in many design patterns. The strategy pattern in particular, promotes composition over inheritance.

In computing and systems design a **loosely coupled system** is one where each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. The notion was introduced into organizational studies by Karl Weick. Sub-areas include the coupling of classes, interfaces, data, and services.

## **Classes should be open for extension, but closed for modification (open-closed principle)**

Classes should be easily extended to incorporate new behavior without modifying existing code. When we accomplish this, our designs are resilient to change and flexible enough to take on new functionality and meet changing requirements.

## **Depend upon abstractions, not concretes**

This is also known more formally as the *Dependency Inversion Principle*. It is similar to the concept of programming to an interface rather than an implementation, but it makes an even stronger statement about abstraction. It suggests that high-level components should not depend on low-level components; they should both depend on abstractions. One example can be visualized by imagining one abstract class inserted between a parent object that contains many child objects. Instead of making the parent refer to the child objects directly, it refers to their abstract base class.

## **Principle of Least Knowledge - interact only with your immediate friends**

For any object in a system, you need to be careful of the number of classes it interacts with and how it comes to interact with those classes.

This principle prevents us from creating systems that have a large number of classes coupled together where changes in one part of the system cascade to other parts. A system with too many dependencies is expensive to maintain and difficult for others to understand.

## **The Hollywood Principle - Don't call us, we'll call you**

Low-level components can participate in a computation, but the high-level components should control when and how. A low-level component should never call a high-level component directly.

## **A class should have only one reason to change**

Assign each responsibility to one class, and only one class.

Modifying code provides opportunities for new problems to arise. If a class has many ways to change, the probability of affecting multiple aspects of a system are increases when the class is changed.

This principle might also be called *Separation of Responsibility*, but we think our chosen heading expresses more. Separating responsibility sounds deceptively simple, but it's can actually be quite hard to do. The human brain is design to classify, organize, and group concepts, so doing the exact opposite can be a bit counter-intuitive.

## Design to avoid Rigidity, Fragility, and Immobility

In his paper, "Dependency of Inversion Principal", Robert C. Martin wrote the following which I think makes a perfect conclusion to this article.

*But there is one set of criteria that I think all engineers will agree with. A piece of software that fulfills its requirements and yet exhibits any or all of the following traits has a bad design.*

*It is hard to change because every change affects too many other parts of the system. (Rigidity)*

*When you make a change, unexpected parts of the system break. (Fragility)*

*It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)*

*Moreover, it would be difficult to demonstrate that a piece of software that exhibits none of those traits, i.e. it is flexible, robust, and reusable, and that also fulfills all its requirements, has a bad design. Thus, we can use these three traits as a way to unambiguously decide if a design is 'good' or 'bad'.*

Keeping all of these principles in mind, as we design, we can create systems that are more easily understood and modified. We can create systems that are less rigid, less fragile, and that are assembled by reusable parts that can be leveraged again and again.

## **Responsibility-driven-design**

Object oriented design is the art of assigning the right responsibilities to the right objects and creating a clear structure with loose coupling and high cohesion. Test driven development (TDD) is a design practice that helps you to achieve this to some extent. TDD drives you towards loosely coupled objects, because too many dependencies will hinder the short test-code-refactor cycles typical for TDD.

Responsibility driven design is an approach that helps you shift focus from object state to interactions and responsibilities.

**Responsibility-driven design expresses the idea that each class should be responsible for handling its own data. Often, when we need to add some new functionality to an application, we need to ask ourselves in which class we should add a method to implement this new function.**

**Which class should be responsible for the task? The answer is that the class that is responsible for storing some data should also be responsible for manipulating it.**

**How well responsibility-driven design is used influences the degree of coupling, and therefore, again, the ease with which an application can be modified or extended.**

**Another aspect of the de-coupling and responsibility principles is that of localizing change.**

**We aim to create a class design that makes later changes easy by localizing the effects of a change. Ideally, only a single class needs to be changed to make a modification.**

**Sometimes several classes need change, but then we aim at this being as few classes as possible. In addition, the changes needed in other classes should be obvious, easy to detect, and easy to carry out.**

**To a large extent, we can achieve this by following good design rules such as using responsibility-driven design and aiming for loose coupling and high cohesion.**

## **Refactoring**

**Refactoring is the activity of restructuring existing classes and methods to adapt them to changed functionality and requirements.**

**Often, in the lifetime of an application, functionality is gradually added. One common effect is that, as a side effect of this, methods and classes slowly grow in length.**

**It is tempting for a maintenance programmer to add some extra code to existing classes or methods. Doing this for some time, however, decreases the degree of cohesion.**

**When more and more code is added to a method or a class, it is likely that at some stage it will represent more than one clearly defined task or entity.**

**Refactoring is the re-thinking and re-designing of class and method structures. Most commonly the effect is that classes are split into two, or that methods are divided into two or more methods.**

**Refactoring can also include the joining of classes or methods into one, but that case is less common.**

We need to understand **encapsulation** to make more sense of some of the design principals we've just covered.

We'll look at getters and setters (also called *Accessors* and *Mutators*)

In short **encapsulation** is a way of keeping access to variables to a minim, it reduces tight coupling.

This is considered a good (and safe) design practice.

**Getters and Setters**  
**also called Accessors and Mutators**

In Java *getters* and *setters*, also known as  
accessors and mutators, are central tools  
In the implementation of 'encapsulation'  
or data hiding – keeping access to variables  
as limited as practically possible

Processing treats classes on other tabs  
as inner classes, so you will not see getters and setters  
used so often, but there are times when you still may use this  
method

In Java **data hiding**, or **encapsulation**, is a central concept. Encapsulation keeps your variables within a minimum degree of visibility. The idea is to protect them from having incorrect or accidental values assigned to them. It also makes debugging easier, as you can more clearly trace where and when the values were assigned.

It is considered good programming style to keep public fields to a minimum, to make your fields(see note\*) private and grant access to those fields selectively by creating **getter** and **setter** methods. You could also, in suitable circumstances omit a setter method so a value cant be changed, this would make it a read only property (there are other ways, such as creating constant variables with the word 'final', eg: **final int WEEKDAYS = 5;** this cant be changed once its been initialized)  
Setter methods can also be used to check that a value is in the correct range for your purposes (eg for a survey or game or whatever)

**\*A Field is a variable that is defined in the body of a class, but outside of any of that class's methods. If they do not have the word static in front of them they can also be called instance variables. The modifier 'private' means that variable is not visible outside of the class, 'public' means it is, 'protected' means it is visible to sub classes (unlike private) but not to outside classes. An instance variable exists to be used when you create an instance of a class (or object)**  
**static** means the variable or method is associated with the class not with an instance of a class, you don't have to create an object to use a static method or address a static variable (but you can access a static method or field from an object instance if you want to.....)

In this way a setter method is validating the data, for example to check that an int is not less than zero or greater than 10. if you had an online survey or scoring mechanism of some sort you could build code like this into the setter:

```
public void setScore(int s)
```

```
{
```

```
  if (s <0)
```

```
    score = 0;
```

```
  else if (s >10)
```

```
    score = 10;
```

```
  else
```

```
    score = s;
```

```
}
```

```
//full code coming up soon
```

We have used lots of built in getter and setter methods in Java and a couple in Processing, for example:

```
g.setColor(Color.red);
setSize(200, 200);
Container contentArea = getContentPane();
contentArea.setBackground(Color.cyan);
slider.getValue();
```

In Processing **set** and **get** methods are used for reading and for writing pixels:

```
get(x, y);
set(x, y, color);
```

Here's a Java example (also in examples folder):

```
class mutantMonster
{
private String color;
private String body;

//constructor for mutantMonster class, these are default values:
public mutantMonster()
{
color = ("orange and pink");
body = ("slimy and cold");
}

public void setMonster(String col, String bod) //setter or 'mutator' method
{
//this could also be broken down into two methods, one for each variable
color = col;
body = bod;
}

public String describeMonster()
//getter or 'accessor' method
{
return("Monster colour is "+ color +", the monster's body is "+ body);
//we could break this down into two methods
}
}
//end of mutantMonster class
```

```
public class monsterData
{
    public static void main(String[] args)
    {
        mutantMonster monsto = new mutantMonster(); //create an instance of mutantMonster class

        //use the setter method of mutantMonster to change values:
        monsto.setMonster("Green","Scaley with tentacles");
        //comment out above to get default values from mutantMonster constructor

        //use accessor or getter method of mutantMonster class to access those values:

        System.out.println(monsto.describeMonster());

        //try to execute these statements - what happens?
        //monsto.color = "purple";
        //mutantMonster.color = "green";
        //System.out.println(monsto.color);
        //System.out.println(mutantMonster.color);
    }
}
```

By declaring the object (or instance) variables 'color' and 'body' to be **private** in the mutantMonster class we protect them from being **directly** changed by **external** program code, we have provided a setter or 'mutator' method to allow the variables to be **manipulated** (mutated!), and a getter (or accessor) method to get access to those variables.

That is **encapsulation**.

We don't have to call our own methods `getSomething()` or `setSomething()` but it helps to call them names that let us and others understand exactly what they do, `describeMonster()` In our example seems more understandable then `getMonster()` but `getMonster()` would be a more conventional naming practice

These names are clear but a bit too awkward would you agree?

```
getMonster_Data()
getMonsterDetails()
```

Its up to you to think of good names in relation to **your own**  
Accessor/mutator methods, the words `get` and `set` are just conventions

Run the previous code and do the suggested experiments -  
and any others that occur to you...

Full version of code that uses a setter method to validate data:

```
class Score
{
private int score;

public void setScore(int s) //setter method with validating algorithm:
{
if (s <0) //if s is less than zero correct value to 0
score = 0;
else if (s >10) //if s is greater than 10 correct value to 10
score = 10;
else
score = s; //otherwise accept value
}

public int getScore()
//getter or 'accessor' method:
{
return(score);
}
}
```

```
public class scoreBoard
{
    public static void main(String[] args)
    {
        Score score1 = new Score();
        score1.setScore(-3);
        System.out.println(score1.getScore());
    }
}
```

## Do you want to know more?

Following are some of the best resources about software design patterns and principles

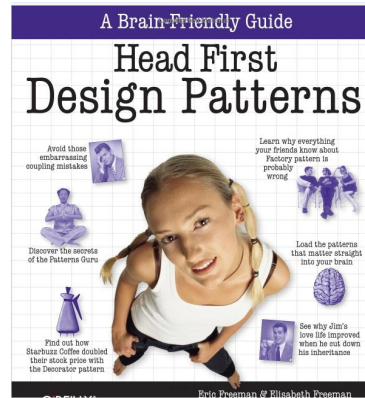
Freeman, Eric & Elisabeth. [Head First Design Patterns](#). O'Reilly Media, Inc., 2004.

[Design Patterns - at Wikipedia](#)

[Patterns Library - at Hillside.net](#)

[Patterns - at the ServerSide.com](#)

[J2EE Patterns Catalog](#)



<https://wiki.base22.com/display/btg/Core+Software+Design+Principles>