ALGONQUIN COLLEGE

MODULE 01: OOP – A CONCEPTUAL OVERVIEW

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 13:15 – 14:15

Friday* 5:00 – 5:30

Email: houtmad@algonquincollege.com

* but confirm beforehand

1.0 OOP – A Conceptual Overview

Object-Oriented Programming (or **OOP**) is programming methodology that applies a divide-and-conquer strategy to software development. In OOP, the programmer considers the composition of the thing being modeled. This involves abstracting the thing in the real world, and then considering the interaction between its components.

Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Booch, G. (1991). *Object-Oriented Design With Applications*. Redwood, California: Benjamin Cummings Publishing Company, pp. 39

1.0 OOP – A Conceptual Overview

Consider the following hardware analogy. A power supply can be thought of as a particular hardware *object*, one component in a larger system, each of which has the following features:

- The object performs one well-defined function, and one function only
- It can be easily replaced if defective
- The inputs and outputs—the public features of the supply—are standardized, both in terms of their voltages and currents, and the physical compatibility of the connectors
- The internal features of the object are hidden; nothing is made available to the user of the power supply that doesn't need to be made available

This is what a well-designed software object should do.

http://www.instructables.com/id/Power-Supply-For-Arduino-power-and-breadboard/?ALLSTEPS

1.0 OOP – A Conceptual Overview

Continuing this analogy: we can idealize each power supply as a 'black box', whose internal configuration and properties need not concern us. All that really matters is the interface to the 'outside world'

Note that while the operation of this device is internally complicated, the simplicity of the interface, combined with the standards agreed upon its manufacturers, helps make each power supply universal, able to be installed in potentially millions of PCs.

1.0 OOP – A Conceptual Overview

With the OOP paradigm, code should work the same way. Each object should have a well-designed interface to the outside world. The actual details of the implementation—the 'insides'—need not concern the user of the code object.

1.0 OOP – A Conceptual Overview

Just as a PC is the result of the interaction of separate hardware components, each computer program results from the interaction between several software objects, each of whose interactions are well-defined and carefully proscribed.

1.0 OOP – A Conceptual Overview

Java is an **Object-Oriented Programming Language** (**OOPL**) in which each code module is packaged into objects. In Object-Oriented Programming, an object's attributes are called its **properties**, while its behaviors are called its **methods**. Properties and methods are sometimes collectively referred to as the **members** of an object

1.0 OOP – A Conceptual Overview

Each object is derived from a **class** in a process called **instantiation**. When properties and methods appear in a *class*, they are called **class fields** and **class methods**. When they appear in an *object*, they become **instance properties** and **instance methods**.

Colloquially, we often refer to just *properties*, *methods*, or *members*, ignoring whether or not they appear in classes or objects.

1.0 OOP – A Conceptual Overview

An **object** is an **instance** of a **class**, and while the two concepts are related, a class is *not* an object. Rather, a class acts like a *template* for an object, something like this:

Just as you can't eat the space *inside* the cookie cutter, in many early OOP languages, you couldn't execute a pure class method; only the object itself contained executable code. (Java blurs this distinction somewhat in its use of **static** members—more on this shortly. For now, the important point is that a cookie cutter can be used as the template for one or more cookies.)

1.0 OOP – A Conceptual Overview

Once the class has been declared, it can be used to instantiate multiple objects:

1.0 OOP – A Conceptual Overview

Each class typically contains one or more special methods, called a **constructor**, that allows each new object to be instantiated with its default properties determined by parameters passed to the constructor

1.0 OOP – A Conceptual Overview

Objects occupy space in memory; once they're no longer needed, they should be deleted from memory. In early OOPLs (like C++) the programmer was responsible for this task; in other languages (like Java), **garbage collection** occurs automatically.

1.0 OOP – A Conceptual Overview

As previously stated, one of the chief benefits of OOP is that it allows the programmer to hide the data stored in an object so that the **clients** of that object can only use it in the ways prescribed by the designer of the class from which the object was derived. To *retrieve* this data, programmers generally provide **getter** (or **accessor**) methods; to *save* the data, **setter** (or **mutator**) methods are employed. Getters and setters thus correspond to the outputs and inputs of our generic object:

This binding together of methods and properties inside objects is referred to as **encapsulation**.

1.0 OOP – A Conceptual Overview

A concept closely related to encapsulation is **data hiding**. Variables which should not be accessible to clients are protected by declaring them **`private`**: their access is prohibited (except via accessor and mutator methods). `private` is an **access modifier** whose purpose is to prevent the client of an object from making unauthorized changes to the attributes of the code.

Nutrition Facts
Serving Size 1 cookie (20g)
Servings per Container 1

Amount per Serving	
Calories 290	Calories from Fat 225
	% Daily Value*
Total Fat 2.5g	14%
Saturated Fat 1g	16%
Trans Fat 0g	
Cholesterol <5mg	21%
Sodium 35mg	12%
Total Carbohydrate 15g	5%
Dietary Fiber 0g	0%
Sugars 8g	
Protein <1g	

`private` information

1.0 OOP – A Conceptual Overview

As a rule, you should *only* make changes to properties using mutator (or setter) methods. You should *only* retrieve property values using accessor (or getter) methods. In other words, getters and setters act as the sole points of contact to private fields.

Nutrition Facts
Serving Size 1 cookie (20g)
Servings per Container 1

Amount per Serving	
Calories 290	Calories from Fat 225
	% Daily Value*
Total Fat 2.5g	14%
Saturated Fat 1g	16%
Trans Fat 0g	
Cholesterol <5mg	21%
Sodium 35mg	12%
Total Carbohydrate 15g	5%
Dietary Fiber 0g	0%
Sugars 8g	
Protein <1g	

private information

This leads to our first statement of best practice:

BP#1:

Never access private fields directly, except in getters and setters

1.0 OOP – A Conceptual Overview

Note: traditionally, the identifier for a setter method starts with the word 'set'; for getters, we prefix the method name with 'get'.

There is one exception to this convention. Methods that return a boolean data type are often prefixed with the word 'has' or 'is', e.g. **has**`Sugar()`, **is**`CertifiedOrganic()`, **is**`NutFree()`, etc. This improves code readability. For example

```
if !(isNutFree())
   System.out.println("Warning: May contain nuts");
```

1.0 OOP – A Conceptual Overview

So getters and setters provide the public interface to 'the outside world', allowing objects to interact with other objects in a safe, allowable fashion. Private members prevent the client of the object from having access to the internal features of the object.

1.0 OOP – A Conceptual Overview

In summary: a class is code module that defines all the essential features that an object will need to function and interact reliably with the other code module in an application. The class's fields will determine the **state** of the object—the features that characterize each instantiated gingerbread man, in our analogy—while the methods determine the object's possible **behaviors**.

While each object presents public methods that the user can call upon to manipulate the object's state, encapsulation ensures that the object's internal mechanisms remain hidden. Thus each class is written as a 'black-box', allowing access to the internal state of an object *only* via public methods.

1.0 OOP – A Conceptual Overview

Consider: the GingerbreadManCookieCutter is but one of a variety of possible cookie cutters designed to instantiate a specific kind of cookie. Each of these cookie cutters shares a common set of functionalities and features, but each 'instantiates' a different cookie object. Each specific cookie cutter class may be considered to have a common set of features inherited from a more general CookieCutter **parent class**. This hypothetical parent class—the presumed 'mother of all cookie cutters'—is referred to as a **superclass**; each **child class**—the specific *kind* of cookie cutter that instantiate different cookie types—is referred to as a **subclass**. For example,

CookieCutter superclass

SnowflakeCookieCutter subclass, GingerbreadManCookieCutter subclass, etc.

1.0 OOP – A Conceptual Overview

This feature, in which one class takes its more-general properties and methods from its superclass, is called **inheritance**.

CookieCutter superclass

SnowflakeCookieCutter subclass, GingerbreadManCookieCutter subclass, etc.

Inheritance is one example of **code reuse**: build and test code once, and then reuse it, rather than recreate it elsewhere.

(Be sure to keep *code reuse* in mind as you go through the following slides.)

1.0 OOP – A Conceptual Overview

The **public** and **protected** members of the superclass get passed down to the subclasses, and hence can be called upon in any instantiated object. For example,

1.0 OOP – A Conceptual Overview

We can add new features simply by adding new properties and methods to any inherited subclass of a superclass (although there are some restrictions, to be dealt with later). These then become available to any object instantiated from that subclass.

CookieCutter superclass

```
public void setSize(…){…}
public int getSize(…){…}
…
```

inheritance

instantiation

inherits, and adds:

```
public void setSize(…){…}
public int getSize(…){…}
…
public void setPoints(…){…}
```

has access to

```
star1.setSize()
star1.getSize()
...
star1.setPoints()
```

inheritance

instantiation

inherits, and adds:

```
public void setSize(…){…}
public int getSize(…){…}
…
public void setShape(…){…}
```

has access to

```
gbman1.setSize()
gbman1.getSize()
...
gbman1.setShape()
```

1.0 OOP – A Conceptual Overview

Additionally, we can **override** any `protected` or `public` members inherited from the superclass, revising them in a subclass to suit our purposes. For example, we can change the size of any newly instantiated gingerbreadman object by overriding the original superclass `setSize()` method in its subclass.

CookieCutter superclass

```
public void setSize(…){…}
public int getSize(…){…}
…
```

inheritance

instantiation

override `setSize()` in subclass

```
@Override
public void setSize(…){…}
public int getSize(…){…}
…
public void setShape(…){…}
```

has access to:

```
gbman2.setSize()
gbman2.getSize()
...
gbman2.setShape()
```

1.0 OOP – A Conceptual Overview

While we can imagine our general CookieCutter superclass as having certain features and behaviours—it has a shape and size, it is made of some material like metal or plastic, it cuts cookies—this class *cannot* be used to make actual cookies, since these are the features of a generic cookie cutter only. It is an **abstract** class.

Only *specific* cookie cutters, like the GingerBreadManCookieCutter, can make actual cookies. These are examples of **concrete** classes, since they can be used to 'instantiate' real cookies. The more general, abstract CookieCutter class, cannot.

This process, in which certain general features are abstracted away from a class into a superclass, is called **abstraction**.

Abstraction and encapsulation are complimentary ideas: abstraction addresses the common observable behaviours of a class, while encapsulation hides the features that should be kept hidden from the user.*

*See: Abstraction VS Information Hiding VS Encapsulation [Webpage] retrieved from http://stackoverflow.com/questions/24626/abstraction-vs-information-hiding-vs-encapsulation, 2015. My definition (above) is a rough paraphrase of Booch, G. (1993). *Object Oriented Analysis and Design with Applications, 2e*. Don Mills, ON: Benjamin Cummings. pp 49, which is cited in the web page just cited near the top of the page.

1.0 OOP – A Conceptual Overview

Furthermore, we can imagine that the abstract CookieCutter superclass is itself descended from a Bakeware class, which itself is descended from a Kitchenware class—and each of these are themselves abstract classes.

Collectively, this derived series of super- and subclasses (whether abstract or concrete) is known as a **class hierarchy**; it describes all the features an instantiated object will inherit based on the properties and methods of its parent superclasses.

In each case, the subclasses inherit the non-private members of the superclass, which then get 'passed down from generation to generation', unless they are overridden in a subclass

Kitchenware class

Bakeware class

CookieCutter class

SnowflakeCookieCutter subclass,
GingerbreadManCookieCutter subclass, etc.

1.0 OOP – A Conceptual Overview

Encapsulation, inheritance, and abstraction are considered to be foundational concepts in OOP.

The fourth and final pillar of the OOP philosophy is **polymorphism**. This feature allows an object of one class to be treated *as if* it were derived from any class higher up in the class hierarchy.

For example, any object derived from the GingerbreadManCookieCutter class could be treated as a member of the CookieCutter class, the Bakeware class, or the Kitchenware class, since, in effect, a GingerbreadManCookieCutter belongs to all of these categories—at different levels of complexity.

Kitchenware class

Bakeware class

CookieCutter class

SnowflakeCookie class,
GingerbreadManCookie class, etc.

1.0 OOP – A Conceptual Overview

What is polymorphism used for? As one possible example, let's say we wish to store an array of cookies. If the array is declared as being of type GingerbreadManCookieCutter, then that array will *only* be able to hold GingerbreadManCookie objects.

1.0 OOP – A Conceptual Overview

But if our array is of the more generic CookieCutter type, then it will be able to hold *any* kind of cookie object whose subclass descends from the CookieCutter superclass. Polymorphism allows us to treat the objects of a subclass (cookies derived from *specific* cookie cutters) as if they were instances of the more *general* CookieCutter class: a gingerbread man cookie is still the product of *a* cookie cutter, regardless of its *specific* features.

1.0 OOP – A Conceptual Overview

One potential complication can occur when objects are used polymorphically: the user of a method or variable may attempt to store the wrong kind of object in an array of a particular type, or pass the wrong kind of object into a method.

For example, say we wish to store cookie *and* donut objects in an array, so we declare the array as being of type Bakeware, which allows us to store all baked goods. But this leaves open the possibility that some user might attempt to store cakes and loaves of bread in the array as well, which probably wasn't our intention.

1.0 OOP – A Conceptual Overview

This problem, in which objects of one type have polymorphic access to the objects (and their methods) of another, perhaps inappropriate type, is called **type safety**. Like code reuse, it forms a major theme in most modern strongly-typed programming languages (of which Java is one), and it will occupy much of our attention later on in this course.

Java uses **generic** declarations to handle this problem. Combined with **interfaces** (a class-like construct), Java allows the designer of a class to limit the range of possible polymorphically-related assignments to just the ones deemed to be acceptable: we should not be able to store baguettes in a tray intended for cookies and biscuits.

(A simpler type safety issue, which you should be familiar with, occurs when we attempt to store a larger numerical data type, like a `long` value, into a narrower type, such as an `int`. The solution, **casting**, is potentially dangerous and should be avoided whenever possible. But it is sometimes necessary.)

1.0 OOP – A Conceptual Overview

You may be asking: where do these superclasses come from? There are two possible sources:

1. Someone else builds them; you import them into your projects using the `import` statement

2. You build and test them yourself, and add them to your Eclipse project

1.0 OOP – A Conceptual Overview

Regardless of a class's parents, each instantiated object gains all the power and functionality built into the methods it inherits from its superclass(es). This applies equally well to simple objects, like our conceptual gingerbread man cookies…

…as well as to more sophisticated graphical objects, like those imported in one or more java libraries, as well as the superclass objects we build into our programs.

```
javafx.application.Application
javafx.stage.Stage
javafx.scene.Scene
```

1.0 OOP – A Conceptual Overview

So every new Java application is a combination of both pre-written library code and user-developed code…

1.0 OOP – A Conceptual Overview

Communications between objects must be carefully circumscribed. So much of the business of programming with OOPLs—and much of the latter half of this course—deals with the interactions *between* objects, rather than the internal composition *of* those objects. Abstraction, polymorphism and generics play key roles in ensuring that the information passed between objects is appropriate.

1.0 OOP – A Conceptual Overview

For simple projects, we're done. But more complicated projects may required multiple code modules, of which this object is merely one component. If so, this component must be able to interact which other objects appropriately once it is instantiated, so it too will need suitable setters and getters to allow it to interface with these other code modules. Really large projects may require hundreds of such components.

1.0 OOP – A Conceptual Overview

Rather than draw 'black boxes' with jig-saw puzzle-like connections between them to represent bespoke classes, we typically use UML class diagrams. Shown below is a sample UML class diagram for a hypothetical class called `Circle`. It has three compartments (boxes), one each for its class name, fields, and methods/ constructors.

Circle
`-radius: double`

`+Circle()`
`+Circle(newRadius: double)`
`+getArea(): double`
`+getPerimeter(): double`
`+setRadius(newRadius: double): void`

This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 324.

1.0 OOP – A Conceptual Overview

A group of classes acting together are typically stored in a **package**. (In Eclipse, a package is really just a subfolder under a project's src directory.) Package names should always be lowercase; this helps distinguish them from classes.

Packages have their own symbol, which looks like a file folder, as indicated below.

1.0 OOP – A Conceptual Overview

Large programs typically contain multiple packages, with special lines both inside and between packages indicating the kinds of connections between classes and other program components.

Shown at right is a small fragment of the UML diagram for a Java-based Pascal compiler*. The complete UML diagram includes several packages and dozens of classes.

*from *Writing Compilers and interpreters: A Software Engineering Approach, 3e*; Ronald Mak, Wiley Publishing, 2009

1.0 OOP – A Conceptual Overview

Of course, for the OOP methodology to work properly, all classes must be coded correctly and tested completely. Failure to do so—to take shortcuts in writing the code, to test code inadequately, or fail to plan ahead—to see the 'big picture'—means that classes and code—and therefore all their derived subclasses and objects—will be hobbled from the start.

1.0 OOP – A Conceptual Overview

To summarize:

- Object-oriented programming is a programming paradigm in which *properties* and *methods* are bundled together into a *class* in a process called *encapsulation*.
- A central feature of OOP is *code reuse*.
- In general, classes must be *instantiated* into *objects* to be executable (although Java has shortcuts around this feature, involving the use of `static` members).
- Special methods known as *getters* and *setters* (or *accessors* and *mutators*) are used to get and set the data stored in objects.
- The four main pillars of OOP are:
  1. *Data hiding* via *encapsulation*. Data that does not need to be accessible to *clients* is kept hidden; at the programmer's discretion, it may be made accessible via getters;
  2. *Inheritance* is a feature that allows a *subclass* to inherit the properties and methods of its *superclass*;
  3. *Abstraction* means that certain common features found in subclasses are abstracted away into abstract superclasses, which cannot be instantiated;
  4. *Polymorphism* means that a class at one level is treated as if it were derived from a class at a different level in the same *class* or *object hierarchy*.

1.0 OOP – A Conceptual Overview

OOP programmers disagree over how many 'pillars of OOP' there actually are—the number varies from three to seven. For example, some programmers view encapsulation as being a part of abstraction (for reasons touched upon earlier), thus reducing the total by one.

(See, for example, http://themoderndeveloper.com/the-modern-developer/back-to-basics-three-or-four-oop-pillars/ or http://www.c4learn.com/cplusplus/cpp-pillars-of-oop/ for discussions on the subject.)

Questions

1. Fill in each blank in the sentences below with one of the words high-lighted in **blue** in this module. (Note: these questions are typical of those found on exams)

   a) An object is the _____ of a class, a process known as _____

   b) A getter is also called a _____, while is setter is known as a _____

   c) Unused objects are cleared from memory via a process known as _____ _____

   d) The binding together of attributes and their behaviours in a class is known as _____

   e) _____ classes can be derived from both abstract and _____ classes

   f) A parent is to a child as a _____class is to a _____class

Questions

2. Consider the following incomplete, hypothetical class hierarchy. Which classes are probably abstract, and which are concrete?

Questions

3. (a) For each of the following images, provide the OOP term from the word cloud below that best describes the relationship between the pictures.

Note: OOP concepts are interrelated, and so any of these questions may have more than one 'almost-right' answer. Try to pick the word(s) that fit the best.

Questions

3. (b) For each of the following images, provide the OOP term from the word cloud below that best describes the relationship between the pictures (con't)

Questions

3. (c) For each of the following images, provide the OOP term from the word cloud below that best describes the relationship between the pictures (con't)

Questions

3. (d) Which of the OOP terms in the word list below best describes what this picture symbolizes?

Questions

3. (e) For each of the following images, provide the OOP term from the word cloud below that best describes the relationship between the pictures (con't)

Questions

3. (f) Which of the OOP terms below best describes what the picture below symbolizes?

Questions

3. (g) Which of the OOP terms below best describes what an ATM *does*?