ALGONQUIN COLLEGE

MODULE 01: OOP – A CONCEPTUAL OVERVIEW

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 14:15 – 15:30

Wednesday (after lecture*)

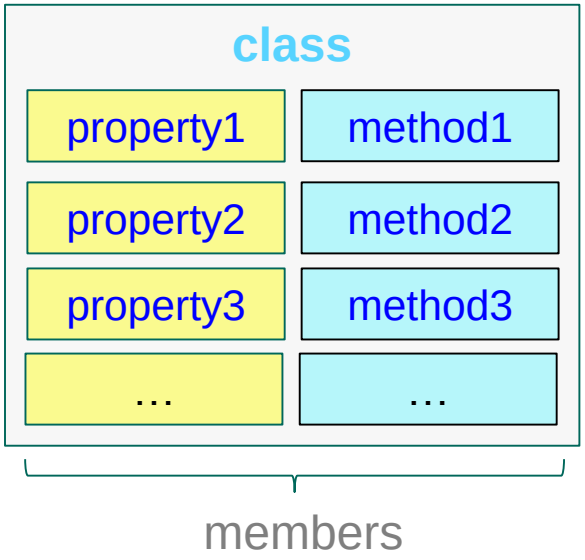
* confirm beforehand

Email: houtmad@algonquincollege.com

1.0 The OOP Model – A Conceptual Overview

Java is an **Object-Oriented Programming Language** (**OOPL**). *Object-Oriented programming* is a programming paradigm in which attributes (called **properties** or, sometimes, **fields**) and the behaviors (called **methods**) that operate on those attributes are packaged into a single entity, called a **class**.

Properties and methods are sometimes collectively referred to as the **members** of a class or object.



1.0 The OOP Model – A Conceptual Overview

Central to the OOP philosophy is the concept of **code reuse**: write the code once, test it thoroughly, and then use it again and again. As you go through the next few slides, keep this feature in mind, since it shows up repeatedly: it is the core concept of object-oriented programming. It will appear again and again in many different forms throughout this course.

1.0 The OOP Model – 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:

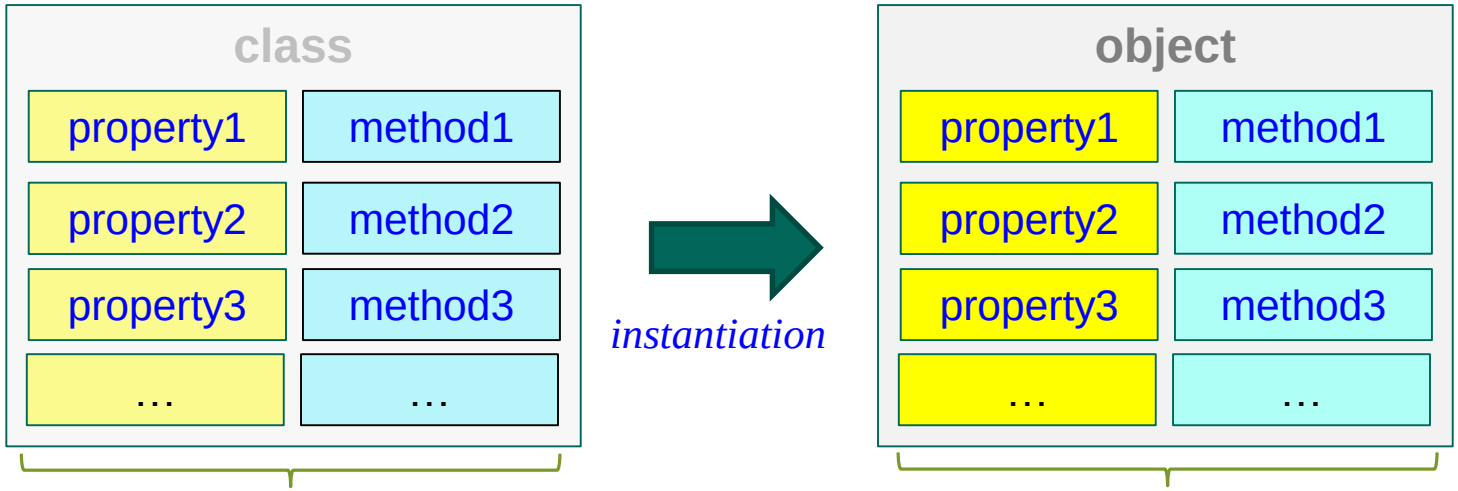
class RAM (Memory) object

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

1.0 The OOP Model – A Conceptual Overview

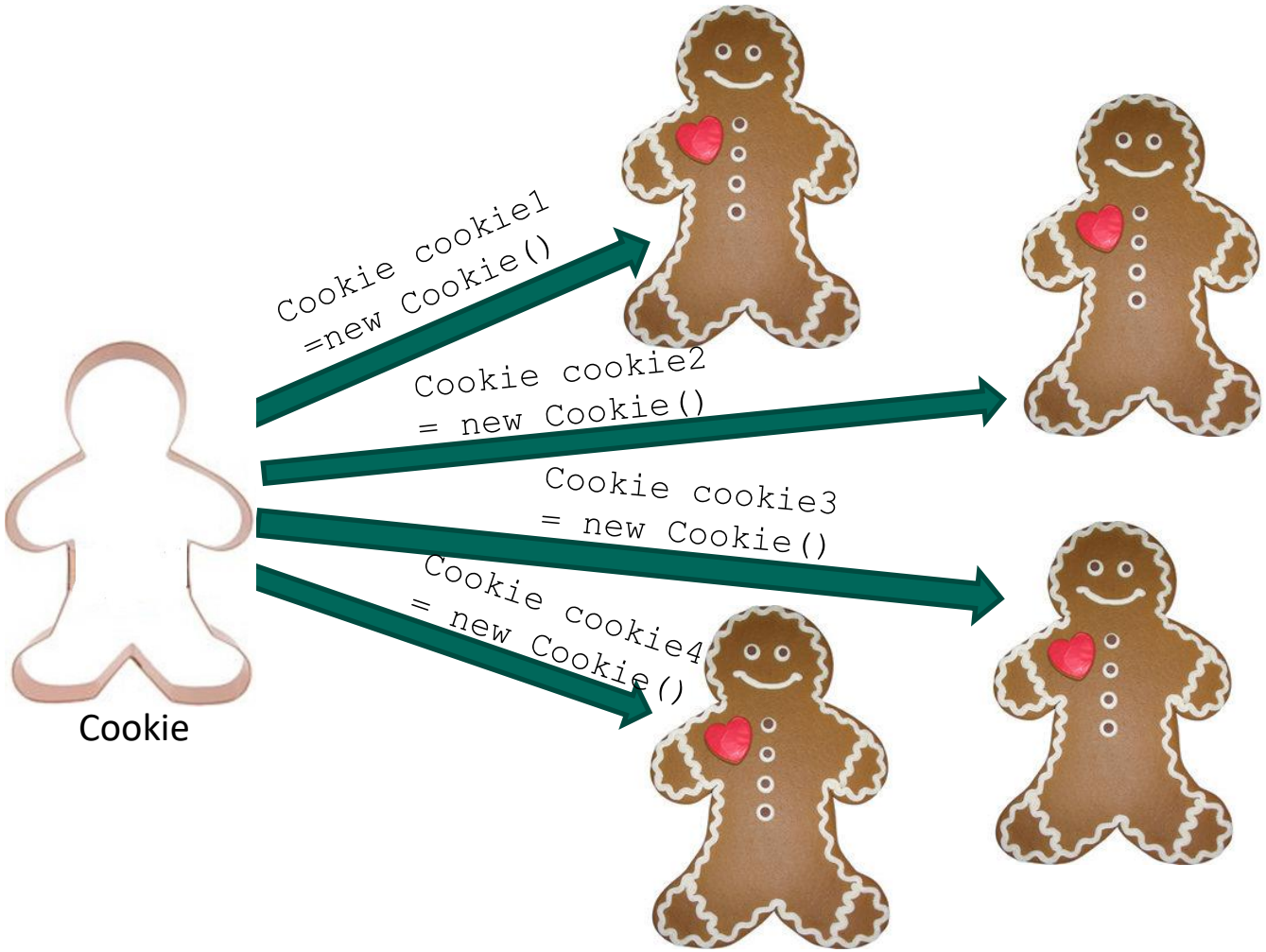
When properties and methods appear in a class, they are called **class fields** and **class methods**. When they appear in an object, they become **instance fields** (or **instance properties**) and **instance methods**.

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



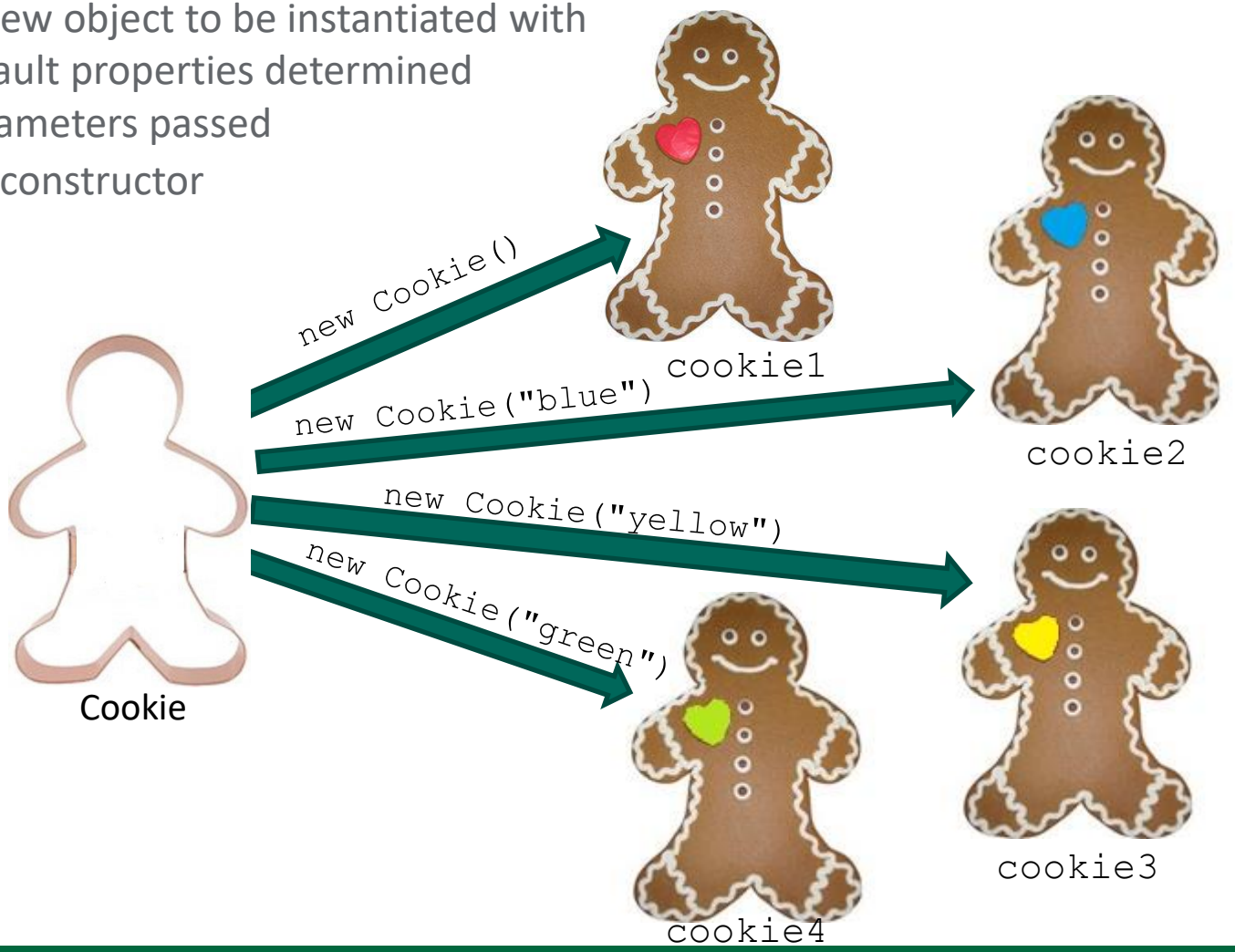
1.0 The OOP Model – A Conceptual Overview

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



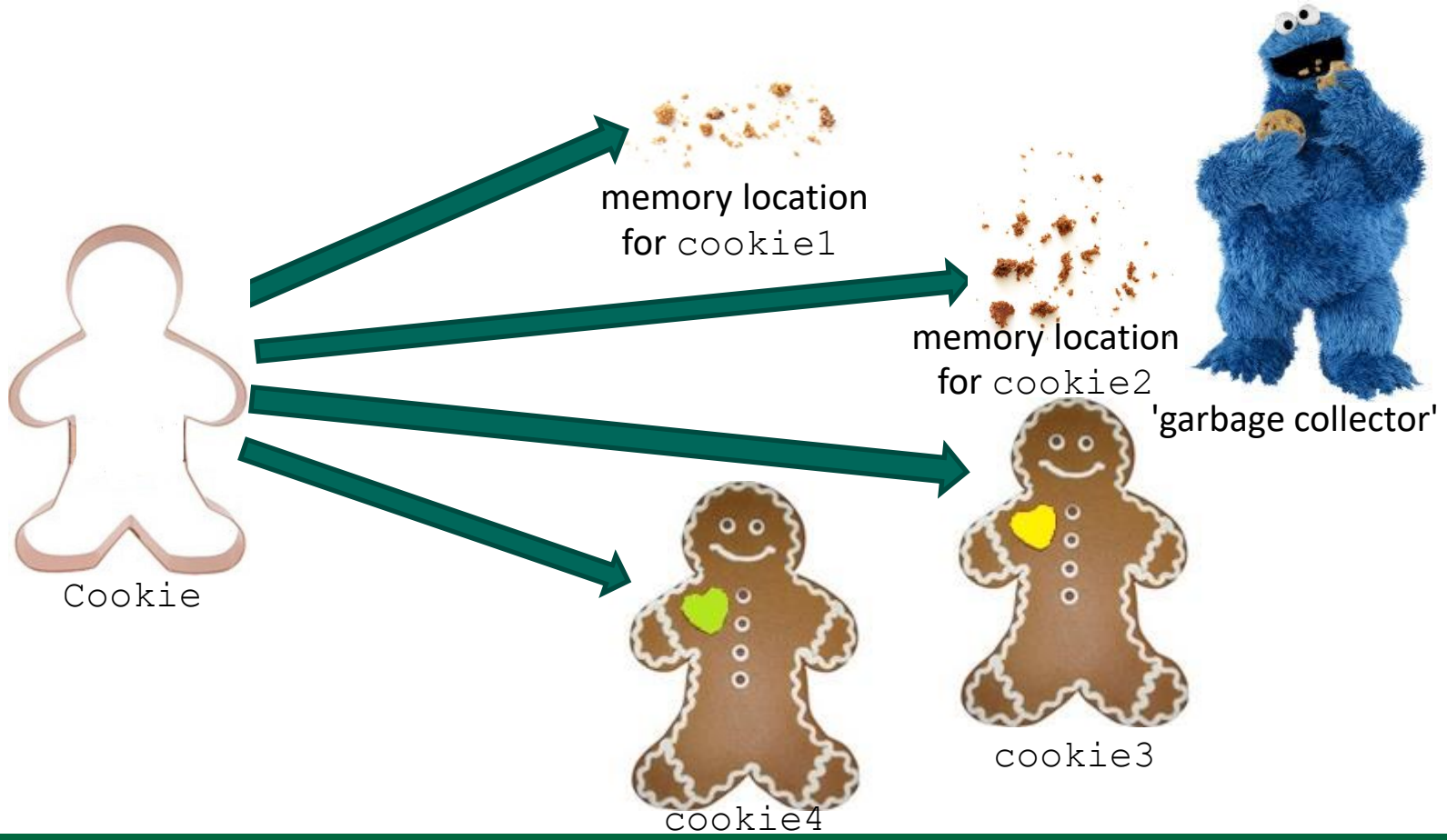
1.0 The OOP Model – A Conceptual Overview

Each class 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 The OOP Model – A Conceptual Overview

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



1.0 The OOP Model – A Conceptual Overview

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.

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

1.0 The OOP Model – 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 possibly 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 or behaviours of the code.

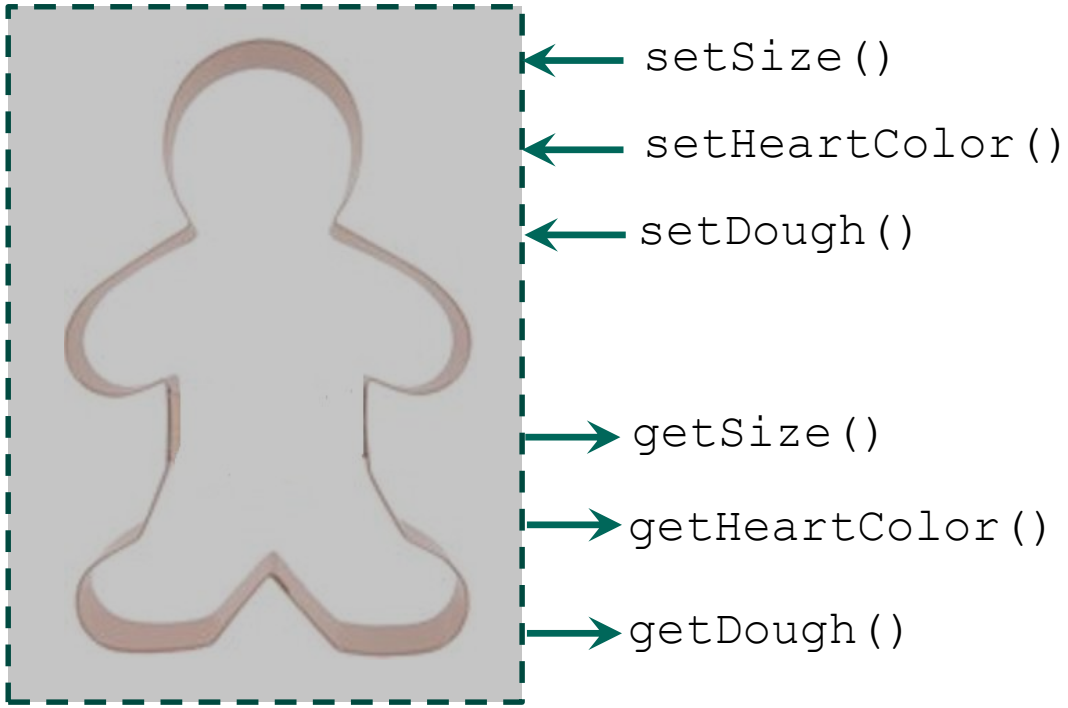
You should only make changes to properties using mutator (or setter) methods. You should only retrieve property values using accessor(or getter) methods.

`private` information (you can see this, but you can't change it)

1.0 The OOP Model – A Conceptual Overview

So a class is a package of code that contains all the essential features that an object will need.

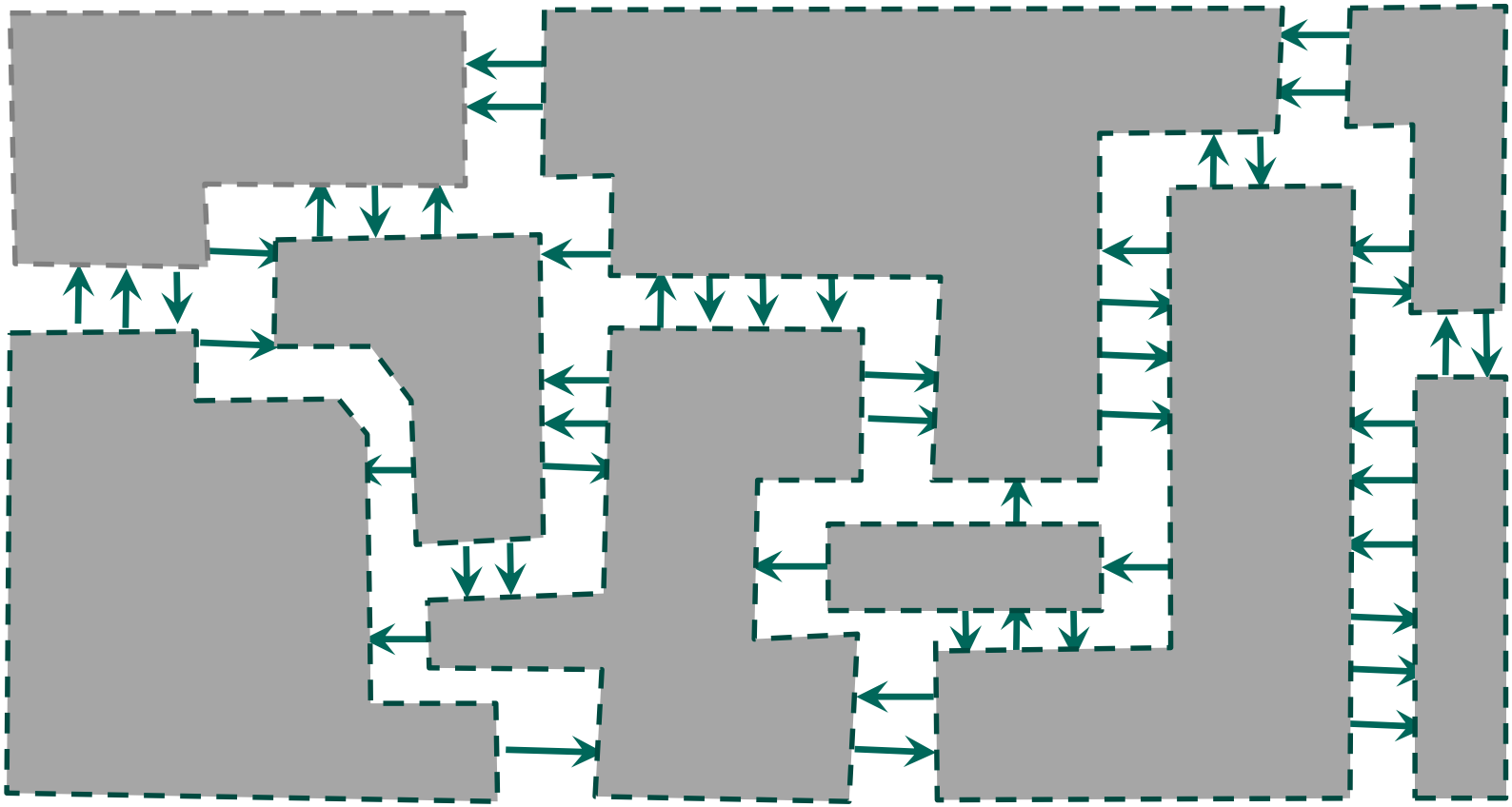
The class's properties will determine the **state** of the object—the features that characterize each instantiated gingerbread man, in our analogy—while the methods determine the object's possible **behavior**.



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 The OOP Model – A Conceptual Overview

A computer program results from the interaction between several objects, each of whose behavior and state is well-defined and carefully proscribed



1.0 The OOP Model – A Conceptual Overview

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

Cookie Cutter
superclass

SnowflakeCookie, GingerbreadManCookie, etc.
subclasses

1.0 The OOP Model – A Conceptual Overview

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

Cookie Cutter
superclass

inheritance

SnowflakeCookie class, GingerbreadManCookie class, etc.

1.0 The OOP Model – 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.

Cookie superclass → *inheritance* →  → *instantiation* → 

```
String shape="Generic";
public String getShape(){
   return shape;
}
public void setShape(shape){
   this.shape = shape;
}
```

```
setShape("star");
```

```
println(cookie1.getShape());
// outputs "Star"
```

Cookie superclass → *inheritance* →  → *instantiation* → 

```
String shape="Generic"
public String getShape(){
   return shape;
}
public void setShape(shape){
   this.shape = shape;
}
```

```
setShape("GingerbreadMan")
```

```
println(cookie2.getShape());
// outputs "GingerbreadMan"
```

1.0 The OOP Model – A Conceptual Overview

We can add new features to any inherited subclass simply by adding new properties and methods in the inherited class.

Cookie superclass → *inheritance* →  → *instantiation* → 

```
String shape="Generic";
public String getShape(){
    return shape;
}
public void setShape(shape){
    this.shape = shape;
}
```

```
setShape("star");
private int points=5;
public int getPoints(){
    return points;
}
```

```
println(cookie1.getShape());
// outputs "Star"
println(cookie1.getPoints());
// outputs 5
```

Cookie superclass → *inheritance* →  → *instantiation* → 

```
String shape="Generic"
public String getShape(){
    return shape;
}
public void setShape(shape){
    this.shape = shape;
}
```

```
setShape("GingerbreadMan")
private String dough="Spelt";
public String getDough(){
    return dough;
}
```

```
println(cookie2.getShape());
// outputs "GingerbreadMan"
println(cookie2.getDough());
// outputs "Spelt"
```

1.0 The OOP Model – A Conceptual Overview

Additionally, we can **override** any of the existing non-`private` (i.e. `protected` or `public`) members inherited from the superclass, revising them in the subclass to suit our purposes. For example, we can change the size of any new 'instantiated' gingerbread man object by overriding the subclass constructor to reset the `size` property in the Cookie superclass.

Cookie superclass


```
private int size;
public int getSize(){…}

public void setSize(int sz){
   size = sz;
}
```

```
@Override
public void setSize(int sz){
   super.setSize(
     getSize() * 20);
}
```

`cookie3.getSize()`

1.0 The OOP Model – A Conceptual Overview

Where do parent classes 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 The OOP Model – A Conceptual Overview

Note that while we can imagine our general Cookie Cutter superclass as having certain features and behaviours—it has a shape and size, 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 can make actual cookies. These are examples of **concrete** classes, since they can be used to 'instantiate' real cookies; the more general, abstract Cookie Cutter 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.*

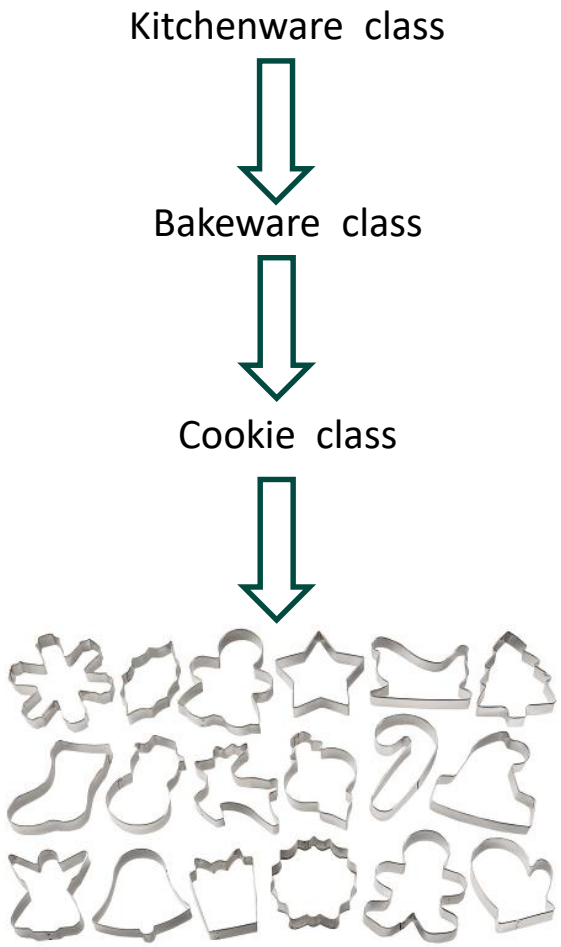
*See: Abstraction VS Information Hiding VS Encapsulation [Webpage] retrieved from http://stackoverflow.com/questions/24626/abstraction-vs-information-hiding-vs-encapsulation, 2015. My definition (above) is a rough paraphrase of Booch, G. (1993). *Object Oriented Analysis and Design with Applications,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

1.0 The OOP Model – A Conceptual Overview

We can imagine that the abstract Cookie Cutter superclass is itself descended from the Bakeware class, which itself is descended from the 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

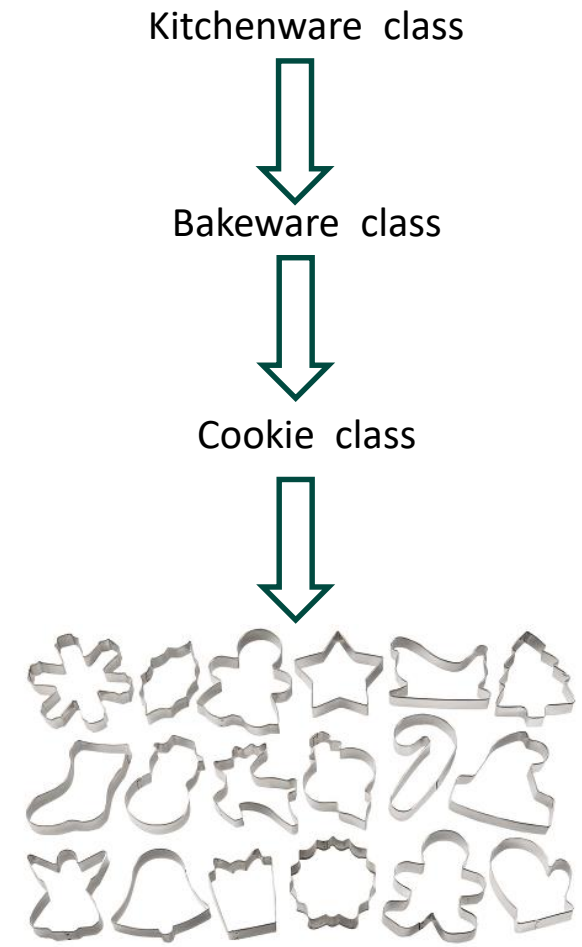


SnowflakeCookie class,
GingerbreadManCookie class, etc.

1.0 The OOP Model – A Conceptual Overview

In addition to encapsulation, inheritance, and abstraction, 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 a class higher up in the class hierarchy.

For example, any object derived from the GingerbreadManCookie class could be treated as if it had been derived from the Cookie Cutter class, the Bakeware class, or the Kitchenware class, since, in effect, a GingerbreadManCookie belongs to all of these categories—at different levels of complexity.



SnowflakeCookie class,
GingerbreadManCookie class, etc.

1.0 The OOP Model – A Conceptual Overview

Let's say we wish to store an array of cookies. If the array is declared as being of type GingerbreadManCookie, then that array will *only* be able to hold gingerbread man cookies. But if our array is of the more generic Cookie Cutter type, then we will be able to hold *any* kind of cookie object whose subclass descended from the Cookie Cutter superclass.

So 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 Cookie class: a gingerbread man cookie is still the product of *a* Cookie Cutter, regardless of its *specific* features.

1.0 The OOP Model – A Conceptual Overview

One potential problem can occur when objects are used in this fashion: the user of a method or variable may attempt to pass the wrong kind of object into that method or variable.

For example, say we wish to store cake and cookie objects in our array. If we declare our array as an array of baked goods then, we can store cake and cookies, but we also leave open the possibility that some user might attempt to store loaves of bread in the array as well, which probably wasn't our intention.

1.0 The OOP Model – A Conceptual Overview

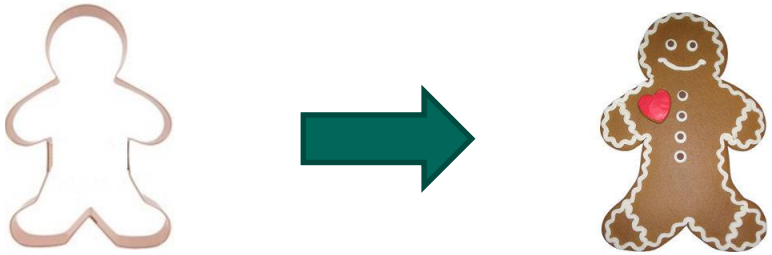
This issue, in which objects of one type have access to the objects (and methods) of another, perhaps inappropriate type, is called **type safety**. Like code reuse, it forms a major theme in most modern OOP languages, and it will occupy much of our attention later on in this course.

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 cake and cookies.

(A simpler type safety issue, which you should be familiar with, occurs when we attempt to store a larger number data type, like a `long` value, into a narrower type, such as an `int`. The solution, **casting**, is problematic and should be avoided wherever possible.)

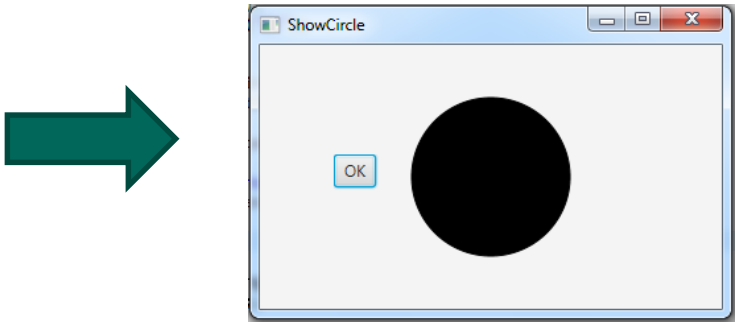
1.0 The OOP Model – A Conceptual Overview

When we import pre-built classes into our projects, the objects we instantiate will gain access to all of the power and functionality built into their methods by the original designer, via inheritance from their superclass(es). This applies equally well to simple objects, like our conceptual gingerbread man cookies…



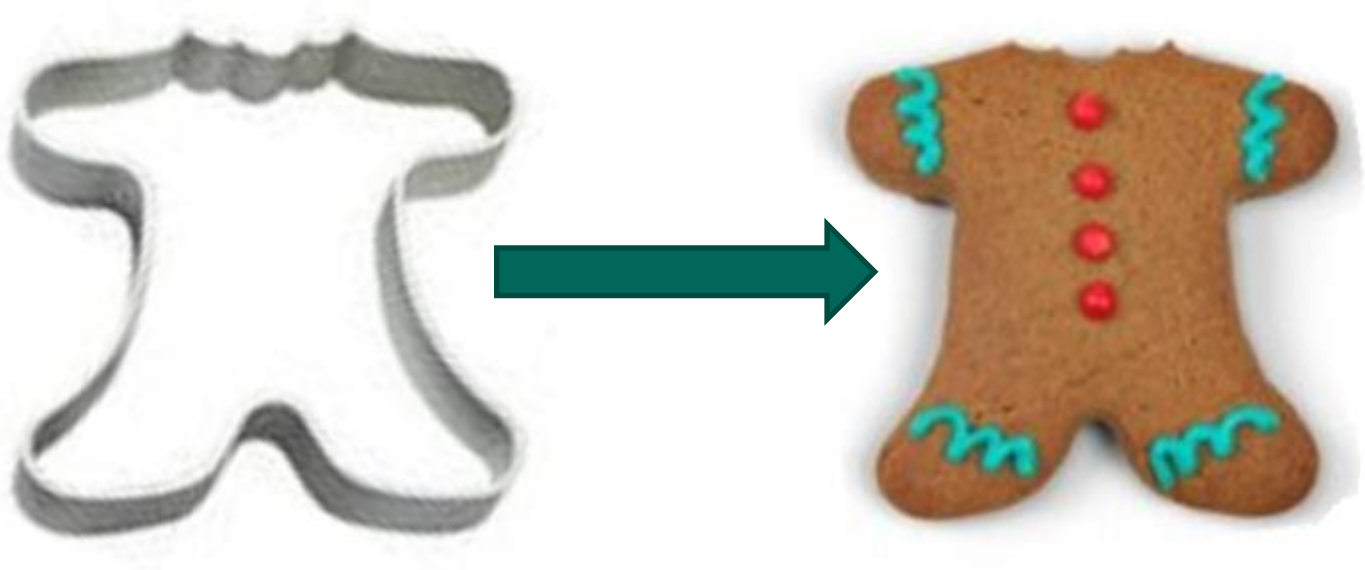
…as well as to more sophisticated graphical objects, like those imported in one or more java libraries

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



1.0 The OOP Model – A Conceptual Overview

Of course, for this 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 The OOP Model – A Conceptual Overview

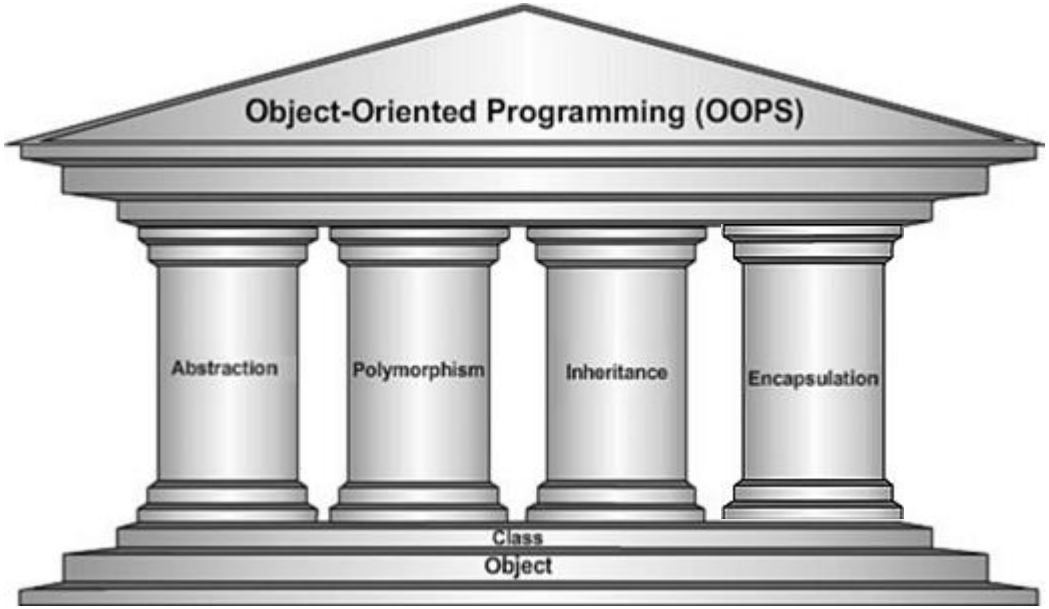
To summarize:

- 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 The OOP Model – Note

OOP programmers disagree over how many 'pillars of OOP' there actually are—the numbers vary from three to seven. For example, some programmers view encapsulation as being a part of abstraction, 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 _____ _____

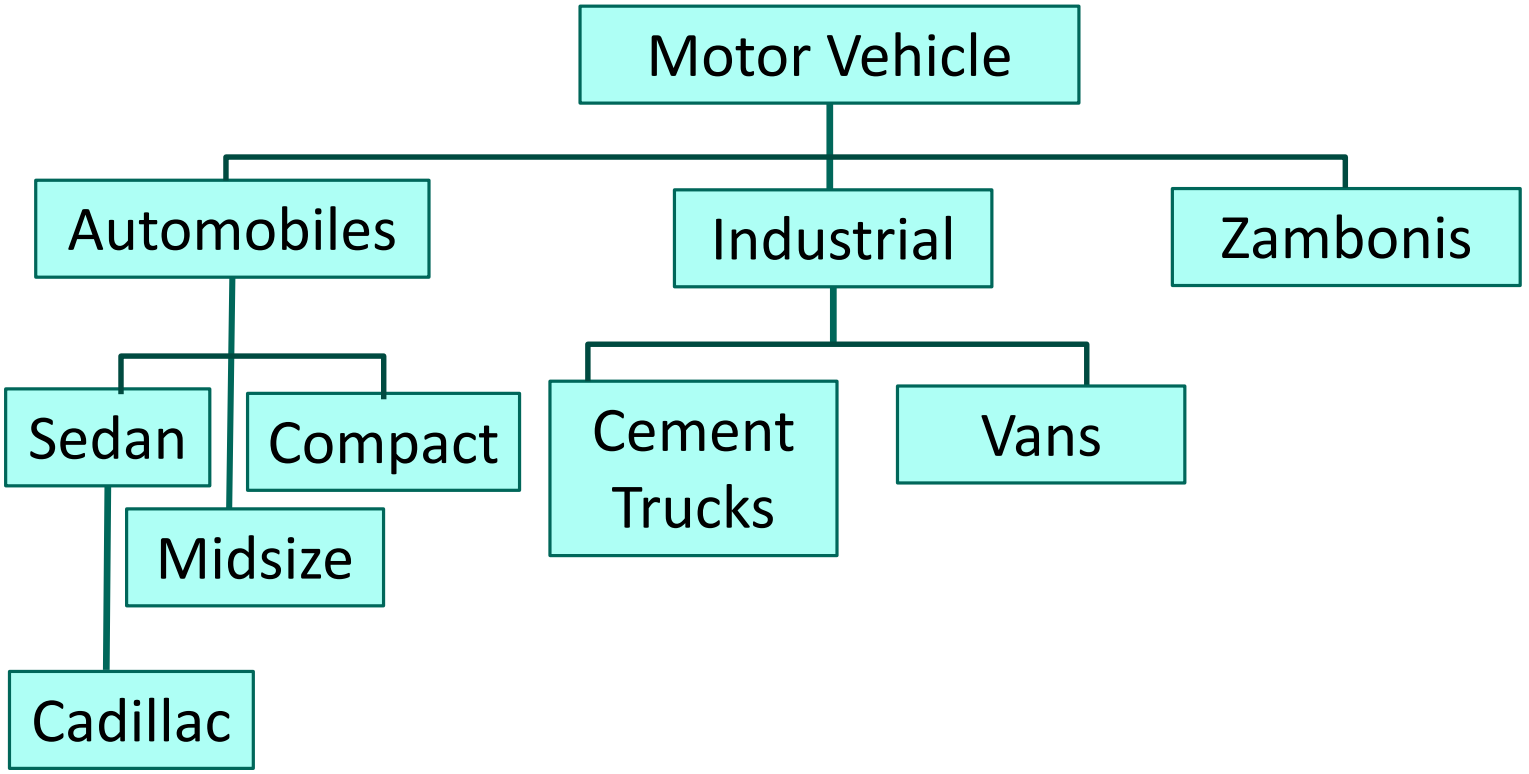
   d) The binding together of attributes and their behaviours in a class is known as _____

   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

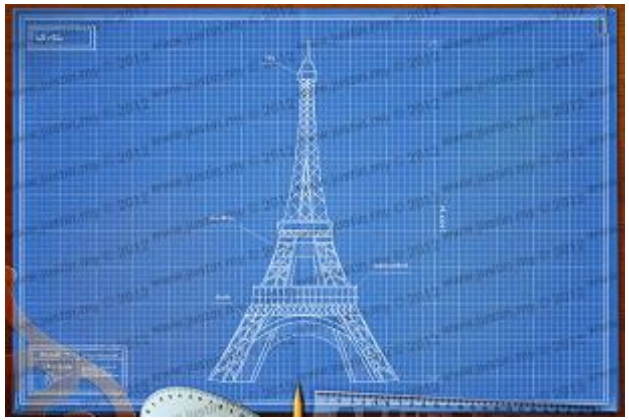
2. Consider the following incomplete, hypothetical class hierarchy. Which classes are probably abstract, and which are concrete?

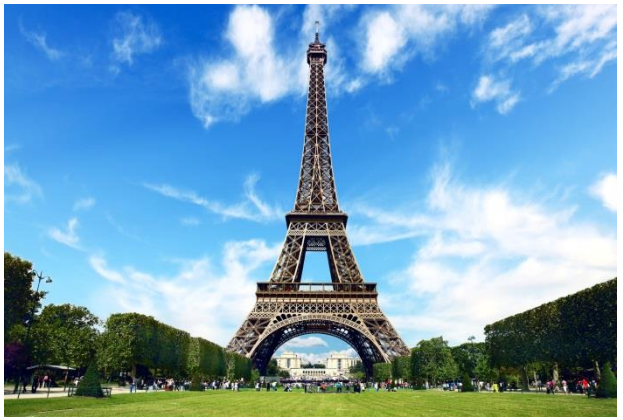


Questions

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 that fits the best.

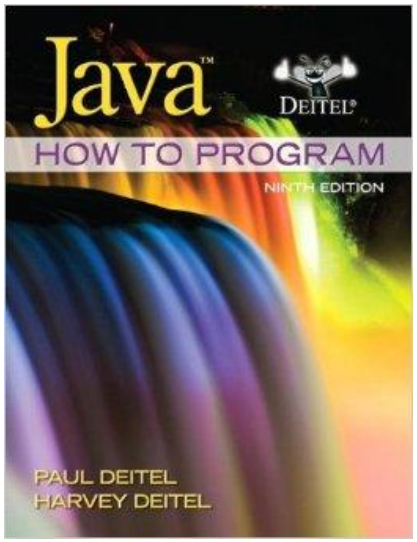


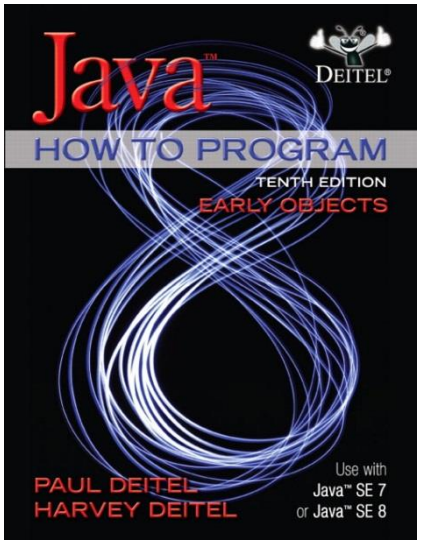


object hierarchy
reuse setters instance
encapsulation constructors
override object methods
instantiate
polymorphism abstract
getters
superclass abstraction
mutators
accessors inheritance
fields
data class OOP hiding
access subclass properties
modifier code

Questions

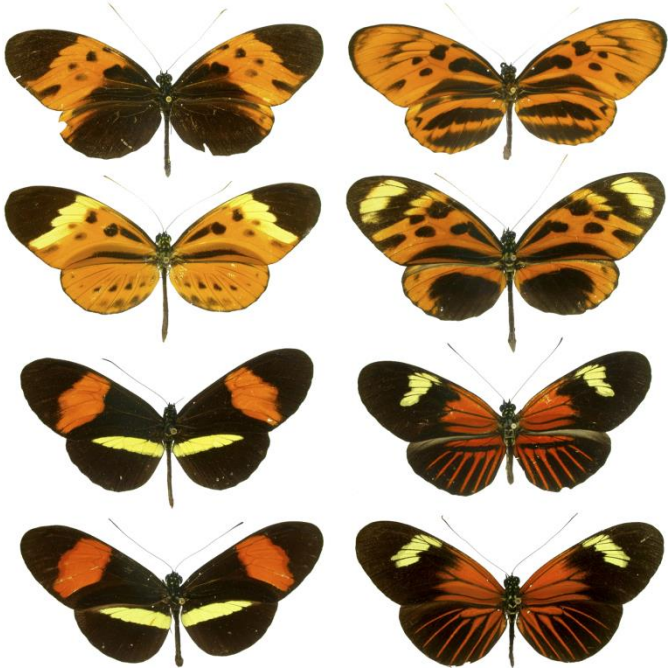
3. (b) For each of the following images, provide the OOP term from the word cloud below that best describes the relationship between the pictures (con't)





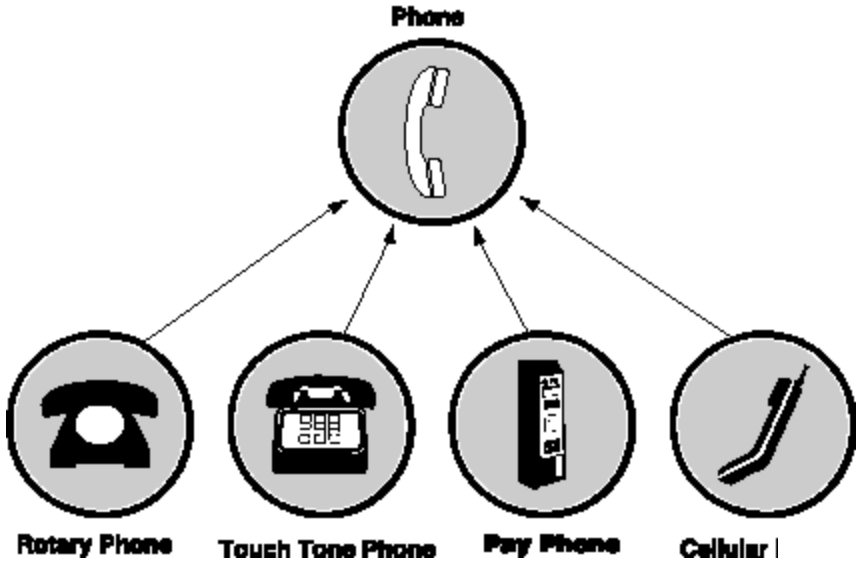
Questions

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

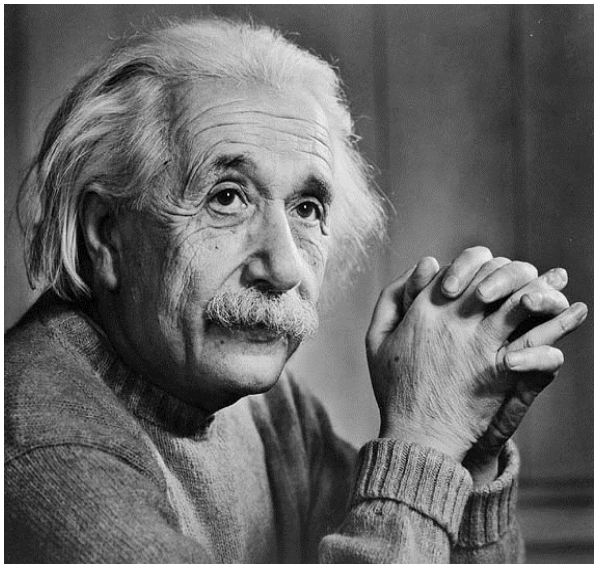
3. (d) Which of the OOP terms in the word list below best describes what this picture symbolizes?

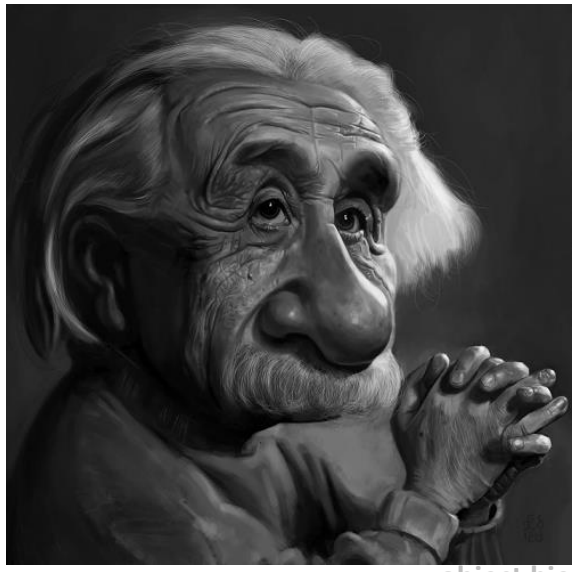


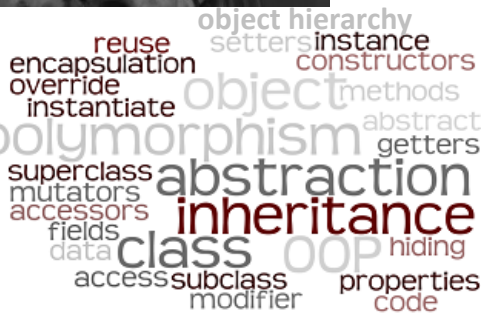
object hierarchy
reuse
setters
instance
encapsulation
constructors
override
object
methods
instantiate
abstract
polymorphism
getters
superclass
abstraction
mutators
accessors
inheritance
fields
data
class
OOP
hiding
access
subclass
properties
modifier
code

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)

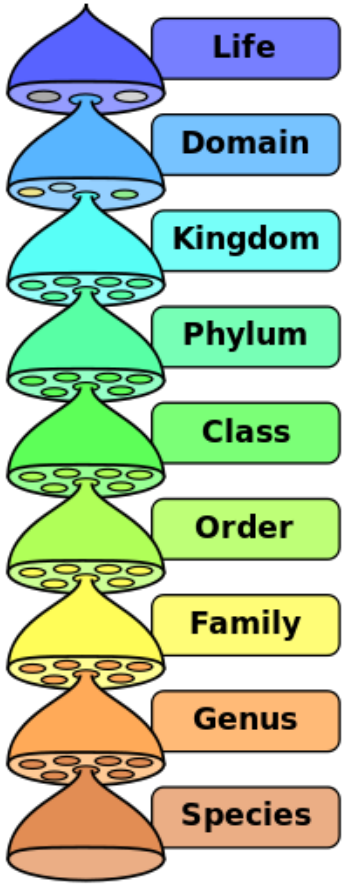






Questions

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*?