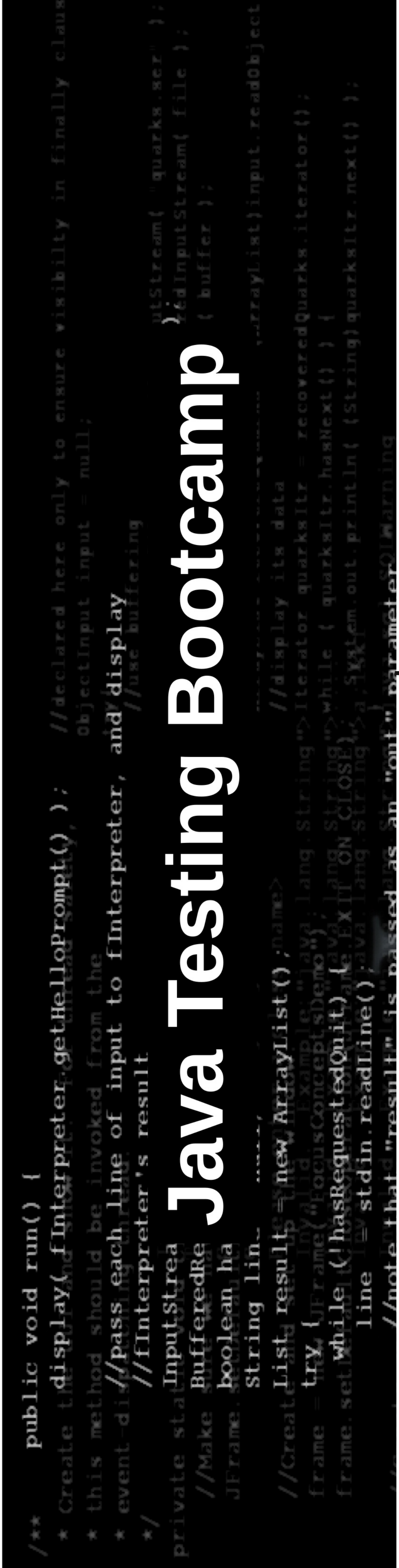


Module Eight

Inheritancel

Before code can be reusable, it must first be usable.
Ralph Johnson

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
C.A.R. Hoare

Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world.
Rebecca Wirfs-Brock

Object-oriented design is the roman numerals of computing.
Rob Pike

Java Bootcamp for Testers

8.1 Introduction

OO Programmers often have a love/hate relationship with inheritance. It makes perfect sense and it seems that it should make programs a lot easier to build, maintain and reuse. However the cost of inheritance is that it often produces “under the hood” complexity that impacts performance, and when inheritance is done incorrectly, it can make code brittle and difficult to modify or extend.

The textbook sections on inheritance ar:

1. In the Java7Notes, all of chapter 5 – especially 5.6 and 5.7
2. In Think Java, Section 14.3
3. The Thinking in Java presentation chapters 6 and 7.

Why lump classes and interfaces together? Because they are both ways of defining “types”. A class is just an implementation of an interface or we can think of an interface as a class that doesn’t yet have an implementation. In other programming languages, notably C++, interfaces are actually just a special kind of abstract class. Java has made the design decision to develop a special kind of abstract class called an interface to allow programmers something called pseudo multiple inheritance – a topic we will pick up in module eight.

6.2 The Object Model

OO has three main concepts, two of them – iconicity and recursive design – were covered in the last module. The third one – the object model – is covered in this module and module eight.

6.2.1 Objects

The concept of an object is central to object-oriented, but the concept can't really be defined clearly because it is too fundamental to the way we think; our brains are wired to interpret the world in terms of objects. But object-oriented is about modeling systems in terms of objects – designing and building objects, and ensuring that we have built our systems and objects correctly (i.e. they do exactly what they are supposed to and nothing else).

The best practical definition of an object belongs to Grady Booch's where he defines an object as

> *Something you can do things to.*

A good test to see if something is an object is to see if it can be measured or evaluated in some way.

The issue of whether objects exist or not independently from the observer is a centuries old ongoing philosophical debate that is entirely beyond this course. However, from a practical and experimental point of view, we know that the objects users think of having an external, objective existence are in fact subjective and may vary from observer to observer depending on their degree of "solidity" but generally there is pretty good agreement about the "things that are out there."

6.2.2 What is an Object?

There is no formal object model like we have for Euclidean Geometry. Instead we have contributions from a number of sources that generally have a common conceptual similarity. Because of the fundamental nature of the concept of an object, trying to come up with a definition for an object is about as useful as trying to come up with definitions for other concepts that refer to wired-in portions of our cognitive systems (like emotion and color).

Here are some samples:

> *An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of Attribute values and their exclusive Services. (synonym: an Instance) Coad and Yourdon,*

> *An object is a visible or tangible thing of relatively stable form; a thing that may be apprehended intellectually; a thing to which though or action is directed. It is a unit of structural and behavioral modularity which has properties. Each object can be characterized by the set of operations that can be performed on it and by the set of operations that it can perform on other objects (either of these two sets may be empty), and by the set of states which it goes through during its lifetime. Each object encloses components, i.e., objects of which it has been constructed, and the operations that can be performed on it, which it makes available for other objects to perform. When these characterizations are accurate, we recognize an object: a self-contained thing which exhibits behavior. Colbert*

> *An object is a person, place, or thing. An object may be physical or conceptual. ... The idea is that an object is a single entity or notion. Each object is a unique individual. An object may be related to or made up of other objects, but each object is unique. Embley*

The object is an Instance of a Class. It is the computer representation of a real world object. The object has Properties as defined in the class of which it is an instance.   IBM

Anything to which a concept, or object type, applies - an instance of a concept or object type. In OOPLs, it is any instance of a class.   Martin and Odell

6.2.3 Describing objects

We can describe what all objects have in common even without having a clear definition of “object.” All objects have:

1. Properties.
2. Behaviors.
3. Identity.
4. Type

We will return to a discussion of these things a bit later on. However, it is important to note at this point that objects that make up a system are independent collections of data and functionality. It is possible that each object has its own internal processing capability and therefore its own version of the class functionality and its own data which may not be the same as the data held by other objects of the same type.

8.2 Inheritance

Inheritance is the converse of abstraction in a sense. We won't try to give some formal definition of inheritance since it is really a formalization of what you do all the time. Instead, we will just describe how it works.

8.2.1 Specialization

If we have a class, lets call it “telephone” and we find that some telephones form a subset of telephones in general because they are telephones with something “extra.” For example we have telephones, the then we have “cellular telephones” which is a special kind of telephone. We also have “watches” and we have special kind of watch called “digital watches.” In the analysis domain, we usually find this specialized class has some sort of adjective to show how it is a specialization of the original class.

The left diagram above shows specialization where we can have an object of type watch, or an object of type digital watch. Specialization says that since digital watch is a type of watch, it does everything a watch does.

The problem is that in the analysis classes, different groups of stakeholders may have different ideas of which class is the specialization. The diagram on the left may be characteristic of the over forty year old crowd, but the under thirty year old group have grown up with digital watches as the norm, so non-digital watches are to them a specialization of the normal digital watch type.

In specialization, we can instantiate objects of the type of the original class (watch) and objects of the type of the specialized class (digital watch).

8.2.2 Generalization

Generalization is where we do the opposite of specialization. Suppose we have digital watches and analog watches as types. We can see that there is something common between them, so we create a new class watch that is a super class of the original two classes.

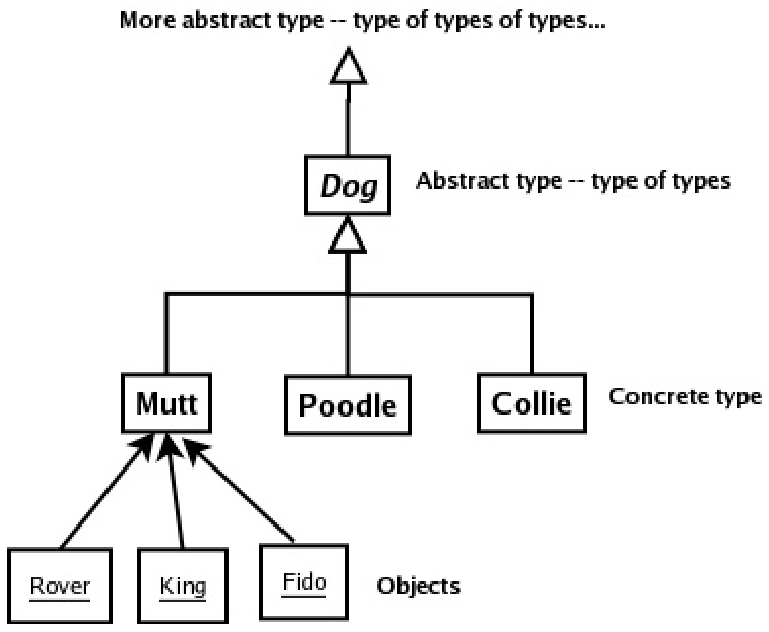
The important thing is that we generalized digital watches and analog watches to create a new type. However, it is now impossible to create an object of type watch since objects of type plain old watch do not exist – a watch has to be either a digital watch or an analog watch. In this case, we call the class watch an abstract class since we cannot instantiate objects directly from it.

8.2.3 Abstract Types

If we think of a class itself as an object (our first axiom) then we can group classes into classes-of-classes. This is the start of our abstraction hierarchy.

How do we know something is an abstract class? Because the class description is generally not complete enough for us to specify a well formed object of that type. Or in plain English, objects of that class just don’t exist.

Consider “bank account”. It is an abstraction of the types “savings account” and “checking account”. Users know this. Try going to a bank and opening a plain bank account -- Insist it be neither a checking or savings account, just a plain bank account. Everyone knows a bank account has to be either a checking or savings account.


More abstract type -- type of types of types...
Dog
Abstract type -- type of types
Mutt
Poodle
Collie
Concrete type
Rover
King
Fido
Objects

This means “bank account” is an abstract class. Notice that the same is true of the types “dog”, “tool” or “food”. When we want to create an object of that type, we have to ask “what kind of tool?” or “what kind of dog?”

8.2.4 Why have Abstract Classes?

Abstract classes, and prototypes in general, are often used to when recording business rules or system knowledge in a form like a script. We tend to store and communicate information as scenarios -- this is called “episodic memory”.

To get money out of an account, a customer goes to an ATM, swipes their card, enters their PIN, selects withdraw, selects the account and amount, then takes their cash and receipt.

The prototypical customer above is an arbitrary object of class customer, as are all of the other objects mentioned. In other words, we use an abstract class as a kind of placeholder in articulating a business rule or a description of a transaction.

When various classes overlap in the ways we looked at earlier for objects (appearance, interaction or relationships) we often capture that commonality by abstracting it out and creating an abstract class, like customer or parent or employee.

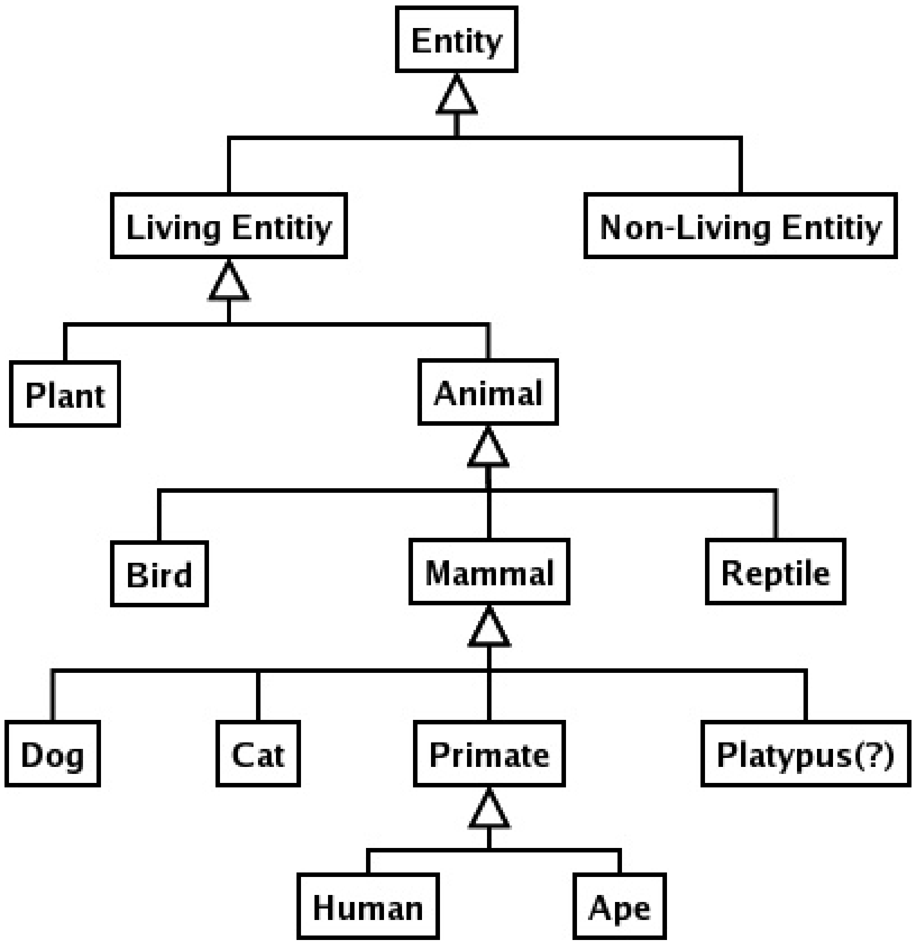
8.2.5 Types or Objects

It is sometimes hard to decide if we are dealing with classes, as in the customer and ATM example above, or actual objects. Remember that we can “do things” to objects. More specifically, an object has properties that have values which we can measure or report on.

In the preceding scenario, we cannot meaningfully answer questions like “What was the customer’s name?” because there is no person object and properties, like name, have to be associated with an object. This comes back to the idea that if you can measure it or evaluate its properties, then it is an object (this is a rule of thumb).

UML humor: "A woman gives birth every 30 seconds in the US. Someone has to find her and stop her." Can you explain the joke above in terms of mixing up classes and objects? For example, the customer in the previous example was not an object but a prototype that stood for the type "customer".

8.2.6 Inheritance Hierarchies



As we create more abstract classes, we eventually construct a class hierarchy, formally called a class decomposition that often is singly rooted. Depending on our orientation, this is either an abstraction hierarchy or an inheritance hierarchy, it just depends on whether we are going up or down the hierarchy.

Each class in the class hierarchy inherits everything from all of the types above it. This is expressed as an "is-a" or "is-a-type-of" relationship.

Notice in this hierarchy a human is a primate which is a mammal. Mammals have are warm-blooded and suckle their young therefor so do humans. Notice that if our hierarchy is wrong (is a platypus is a mammal?), we can make inheritance errors; like assuming the platypus gives birth to live young.

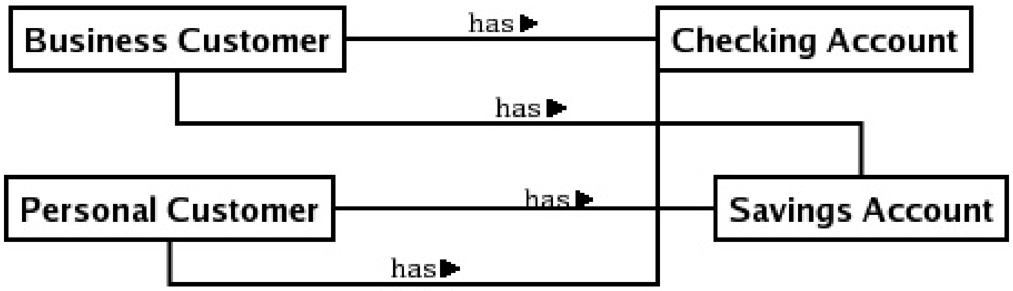
This is something you do all the time – this is just a formalization of your built-in way of organizing the world around you. In OO terminology, primate is a superclass or base class for human and human is a subclass or derived class of primate.

In the diagram above, we have used the standard UML notation to indicate generalization. Specifically, human and ape are generalized (or abstracted) to create the abstract class primate. _Similarly, we generalize dog, cat, primate and platypus to create the abstract class mammal. In any class diagram showing generalization, only the lowest level can possibly be concrete classes – i.e. classes we can create instances from.

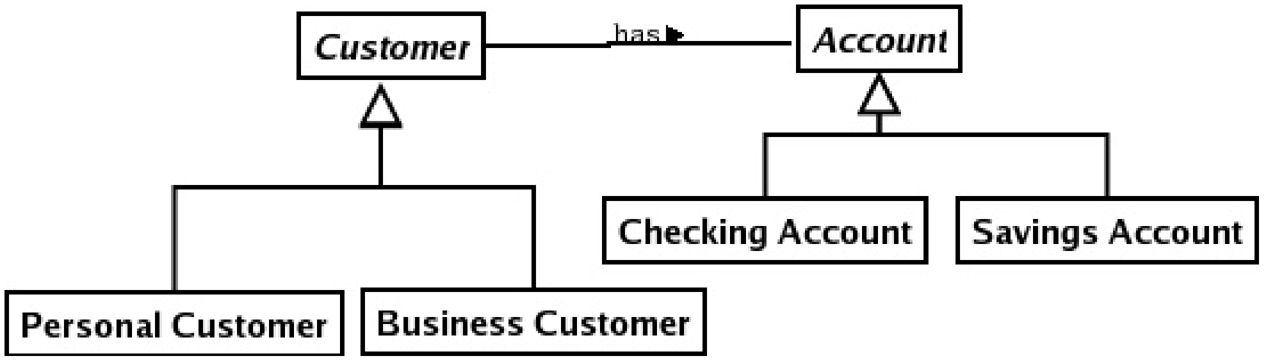
8.2.7 Why Use Generalization?

While the reasons for using generalization in object-oriented in general are beyond the scope of this module, we can state that generalization is used to make our models simpler. How does this work? Generally we use often abstract based on similarity of relationships.

Consider the first diagram below. We have two kinds of customer and two kinds of bank accounts. The diagram shows a lot of association lines.

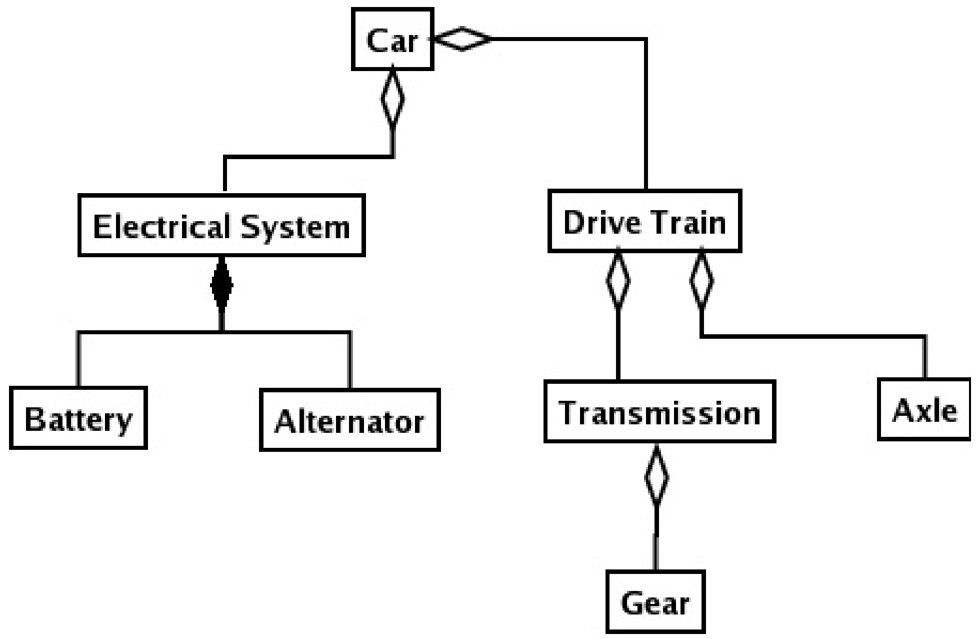


In the second diagram, we see the same model but with two abstract classes introduced to capture the commonality or relationships.



8.2.8 Aggregations and Compositions

The other kind of decomposition is called object decomposition or a part-of relationship. The UML term for this is aggregation. In the graphic below we show a portion of an object decomposition of a car.   Notice that a transmission is part-of a drive train which is part-of a car.



Here is a typical object decomposition. Note that this is a “part-of” hierarchy and so this is an aggregation tree, not an inheritance tree.

But “car” is a class (and an abstract one to boot – British humor)! You had no difficulty understanding the object decomposition for a car even though it was a class rather than an object. Now replace “car” with “motor vehicle”, we have a more abstract decomposition from which our object decomposition of “car” is inferable through inheritance. This is called orthogonality; the use of the class and object decomposition at the same time.

An aggregation can be homogeneous or heterogeneous. A homogeneous aggregation is when all the parts are of the same type - like a list of names. A heterogeneous aggregation is when the parts are of different types, like the car example above.

8.2.9 Composition versus Collection

Aggregations are classified on a scale of how tightly bound the parts are to the container.

At one pole is a collection or loose aggregation.   For example, we might say that a company is an aggregation of employees, since a person can be an employee in more than one company at a time, and the employee has a continued existence even if the company disappears. This example illustrates a couple of the questions associated with determining the degree of binding between an object and its container.

Can an object belong to more than one collection at a time?

Can the object exist independent of the container?

At the other pole is a composition or whole-part relationship. For example, we might say that you are a composition of internal organs, because each organ can be contained in only one person and the organs generally do not have an independent existence after the person container dies. The binding between the object and its container is much tighter in a composition.

So how do we decide that something is a composition or aggregation? It is a matter of how the modeler perceives the relationship. For example, you would probably see yourself as existing in a composition relationship with your internal organs while an organ transplant clinic might see the relationship as a bit looser.

In the previous example, we have modeled electrical system as a whole containing the parts battery and alternator. Our justification (remember as a modeler we make this choice) is that an electrical system is a name for a group of components working together – without the parts, there is no system. Perhaps you can come up with a reason to not use composition but to use aggregation instead.

Generally, we use aggregation (the open diamond). We tend to only use composition when we want to emphasize the whole-part nature of the relationship.