Introduction to Objects and UML

7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111
303-302-5234 | 800-292-3716
SkillDistillery.com

Introductions to Objects and UML

Student Workbook

Introduction to Objects and UML

Lynwood Wilson

Published by ITCourseware, LLC., 7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111

Contributing Authors: John McAlister, Jamie Romero, Rick Sussenbach, and Rob Seitz.

Editors: Danielle Hopkins and Jan Waleri

Editorial Assistant: Ginny Jaranowski

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, offering comments, suggestions, criticisms, and insights.

Contents

CHAPTER 1 - COURSE INTRODUCTION

Course Objectives

- Define fundamental Object-Oriented concepts such as abstraction, encapsulation, inheritance, and polymorphism.
- Model classes and static relationships between them using class diagrams.

Course Overview

- **Audience**: Programmers who are new to Object-Oriented techniques.
- **Prerequisites**: Experience programming in a non-OO language.

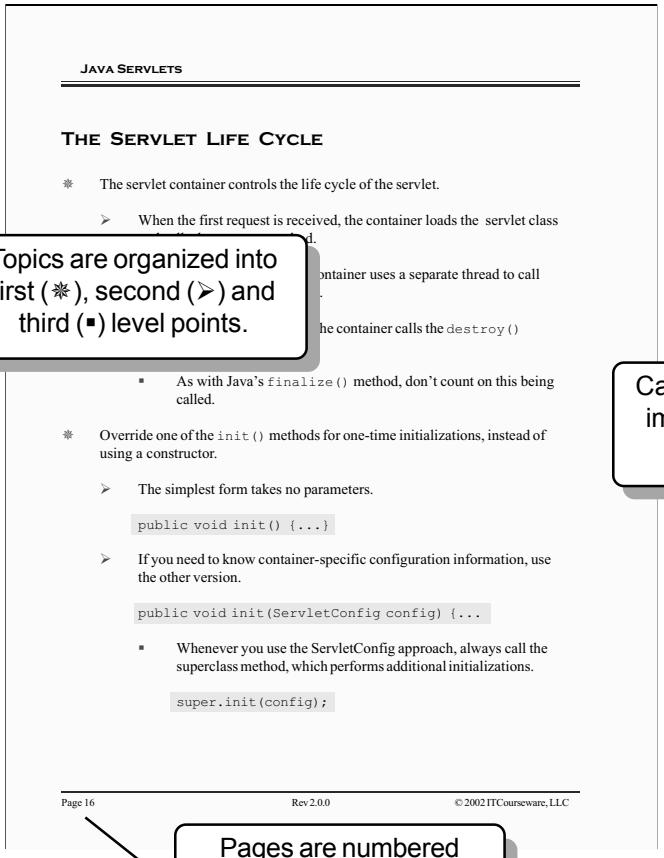
USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as online if you need a little help.

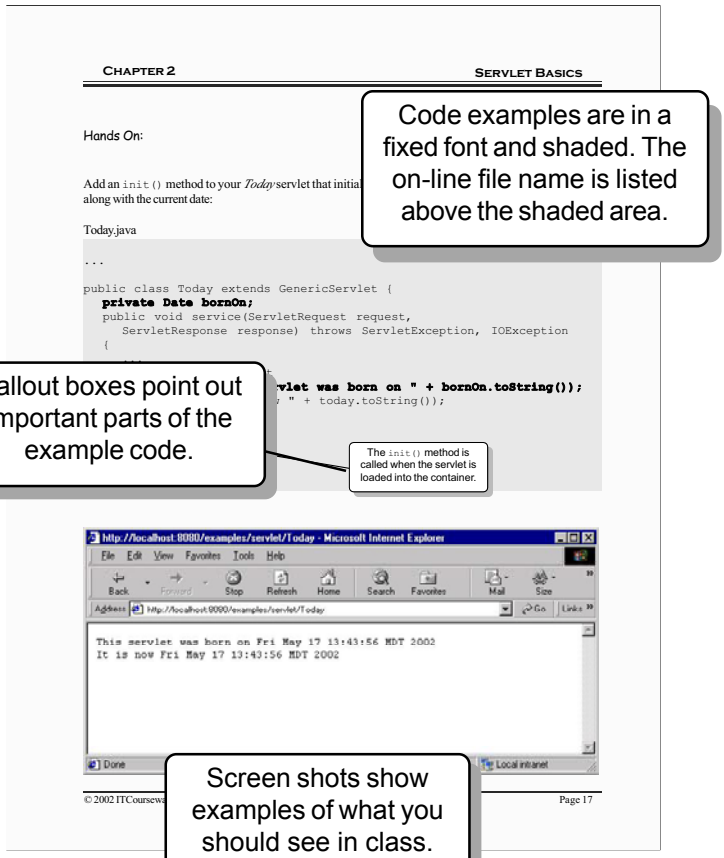
The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.



Topics are organized into first (❋), second (➢) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.



Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

Suggested References

Booch, Grady, James Rumbaugh and Ivar Jacobson. 2005. ***The Unified Modeling Language User Guide, Second Edition***. Addison-Wesley, Reading, MA. ISBN 0321267974.

Fowler, Martin, et al. 1999. ***Refactoring: Improving the Design of Existing Code***. Addison-Wesley, Reading, MA. ISBN 0201485672.

Fowler, Martin. 2003. ***UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition***. Addison-Wesley, Reading, MA. ISBN 0321193687.

Gamma, Erich, et al. 1995. ***Design Patterns: Elements of Reusable Object-Oriented Software***. Addison-Wesley, Reading, MA. ISBN 0201633612.

Hunt, Andrew and David Thomas. 1999. ***The Pragmatic Programmer: From Journeyman to Master***. Addison-Wesley, Reading, MA. ISBN 020161622X.

Larman, Craig. 2004. ***Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131489062.

McConnell, Steve. 2004. ***Code Complete: A Practical Handbook of Software Construction, Second Edition***. Microsoft Press, Redmond, WA. ISBN 0735619670.

McLaughlin, Brett D., Gary Pollice and David West. 2006. ***Head First Object Oriented Analysis and Design***. O'Reilly Media. Sebastopol, CA. ISBN 0596008678.

Miles, Russell and Kim Hamilton. 2006. ***Learning UML 2.0***. O'Reilly Media, Sebastopol, CA. ISBN 0596009828.

Pilone, Dan and Neil Pitman. 2005. ***UML 2.0 in a Nutshell***. O'Reilly Media, Sebastopol, CA. ISBN 0596007957.

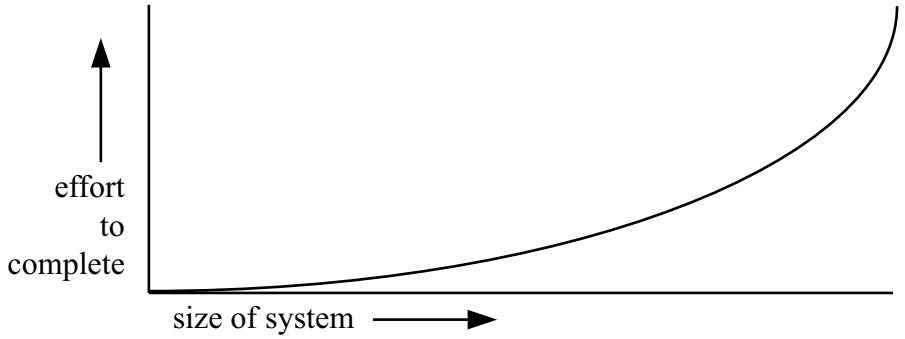
CHAPTER 2 - INTRODUCTION TO ANALYSIS AND DESIGN

OBJECTIVES

- Identify essential problems and tasks of software development.
- Describe basic concepts of modularity and abstraction.
- Outline the concepts of Objects and Object-Oriented Programming.

WHY IS PROGRAMMING HARD?

- It's complicated.
  - Computer programs are among the most complex things people build.
  - Most people can only think of 7 ± 2 things at a time.
- It gets more complicated as the system gets bigger.
  - Why is the curve of *effort vs. size* exponential?
    - More communication links among programmers, designers, analysts, clients, etc.
    - More communication links among modules in the system.
  - The most efficient software project is a single programmer working on a program no one else will use. There's no communication.



- Modularity helps to flatten this curve.
  - With good design and abstraction you can work on a module — a part of the program — as though it were a single small program, and thus stay toward the left end of the above graph.
  - This works even when it's a high-level module that uses several low-level modules, if you properly define and constrain the interfaces.

Miller, G. A., "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psychological Review*, Vol. 63, March 1956, pp. 81 - 97. Miller's research showed that most people can only hold seven plus or minus two things in working memory at a time. Think about it when you are designing menus. Most of our models and drawings should contain no more than nine different top-level artifacts.

Blaise Pascal, they say, once closed a letter by saying, "I'm sorry this letter is so long. I didn't have time to make it shorter." Most of what we create will benefit from taking the time to make it shorter. Text, models, and code alike are clearer and communicate better if we take the time to make them concise, precise, and elegant. Take the time.

You have a client even if you aren't a consultant or a contractor. Your client is the person who manages the group that needs the system you are working on. Often the client is the person who asked for the work, or the person who pays for it or whose division pays for it. Usually the client is the person who knows best what the top-level requirements are. The users are important, and your system must satisfy them, but they often do not know all the needs of the business.

The Tasks of Software Development

- Figure out what problem to solve or what system to build.
  - Analysis
- Build the system to solve the problem.
  - Design
  - Implementation / Programming
- Analysis is harder.
  - Most of the problem is communication: communication with the computer and communication with people. The computer is easier, in spite of (or perhaps because of) being so literal and requiring perfection in each detail.
- What tools do we use to manage complexity and help with communication? (Not just in software, but everywhere.)
  - Modules — Break a job into simpler components such that if we complete the components the job will be done.
  - Models — Represent the problem, the solution, and their component parts in such a way as to enable us to work with the important aspects and ignore the rest (abstraction).
  - Formal Process — Organize the work so that we do everything important with a minimum of non-productive effort.

We divide the analysis into two parts: Domain Analysis, and Requirements or Specification Analysis.

Domain Analysis is finding out about the business and its processes. Building a common vocabulary with the domain people, users, clients. Understanding the context within which our proposed system must operate. And if there is an existing system that ours is to replace, we should understand that as well.

Requirements or *Specification Analysis* describes the system that will solve the client's problems, characterizing it in such a way and to such a depth that if we meet the specification we satisfy the client.

It's not possible to perform either of these perfectly or completely. This is one reason we must deal with change throughout the process, as we discover missing, incomplete, inconsistent, or erroneous specifications.

MODULES

- A *module* is a part of a program or model that can be considered as an entity separate from the rest.
  - A module has a purpose.
  - A module has a specified interface through which it interacts with the rest of the system.
  - A module is abstract. It hides its implementation, the details of its operation, from the rest of the system.
- A module should have high cohesion.
  - It should do one thing, have a single responsibility, at its level of abstraction.
- A module should have low coupling.
  - It should be a black box. The modules that use it need not understand its internal operation.
  - Its external interface should be simple, narrow, and elegant.
- The kind of module we will be most interested in during this course is the object.
  - We will also see higher-level modules that contain multiple objects.
  - Objects contain attributes (data) and methods (functions), and these are modules, too.

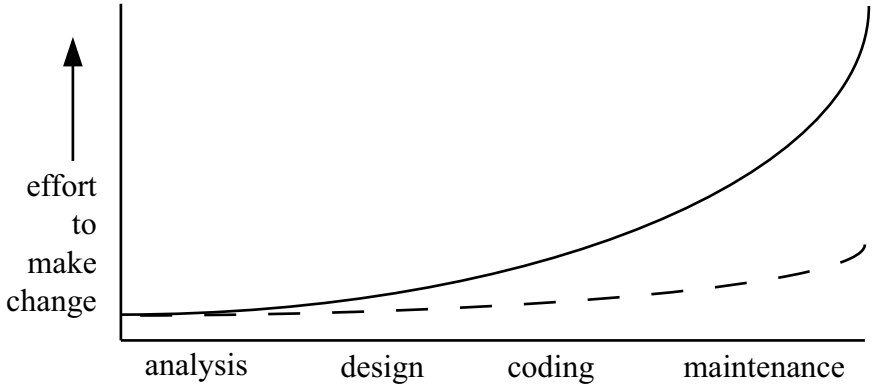
Objects

- Objects are better modules.
- Objects give us more abstraction, better modularity, more flexibility.
  - Now it's modularity of both function and data.
  - We can encapsulate more in a module and enforce the encapsulation in ways that we could not before.
  - The modules are even further from the machine, closer to human thinking.
  - The modules can represent artifacts from the problem and the problem domain, from the world of the people rather than the world of the computer.
- Before OOP we thought first of the operation: What's it do?
  - After we worked that part out, sometimes a long time later, we thought about the data: What's it do it to?
  - In OOP the data gets at least equal attention.
- Each of these advances in abstraction, modeling, and modularity gave us the power to build larger systems with less effort by managing communication problems.
- Objects help us to program the way we actually work in the real world instead of the way we worked in school.

Anthropomorphism in designing OO models and programs is encouraged. In fact, it's one of the advantages of objects. We all know that our programs don't have tiny people inside doing the work, but it's often useful to think of modules as having desires, responsibilities, mental state similar to that of a human.

CHANGE

- Specifications change.
  - This can cause poor communication between us and them.
  - They don't know what they need until we give them what they say they want.
  - Specifications change as we (and they) learn more.
  - Specifications change as the business domain changes.
  - The later we change, the more it costs.



- The solid line above is the classical curve of effort to change the system vs. the point in the development cycle at which we begin. (The same graph describes the cost of fixing a bug vs. the length of time it went undiscovered.)
  - The solid graph is based on data from before OOP.
  - If we design and build good objects and maintain the structure of the system every time we touch it we can flatten this curve into something more like the dashed line.
  - It will always cost more to make a change as time passes, but we can keep the curve from getting so steep.

- OOP and OOD help us to do a better job in the real world, working with change instead of fighting it or pretending we can control it.

We have to handle vague and changing requirements. We cannot force our clients to work or think the way we want them to. And if we could it still wouldn't be the right thing for their businesses or ours. The world is vague and constantly changing, and the rate of change is increasing.

OO helps. If you do a good job at the object level, it's much easier to change the program later when you know more. We want to flatten that curve.

CHAPTER 3 - OBJECTS

OBJECTIVES

- Describe the basic concepts of Object-Oriented Programming.
- Explain abstraction and encapsulation, and how they help us design and write programs.
- Create classes with attributes and methods.
- Describe the difference between instance and class scoped attributes.

ENCAPSULATION

- Encapsulation is the most important feature of Object-Oriented Programming (OOP). Without it, the other big ideas (inheritance and polymorphism) are useless.
  - Note that encapsulation is not new to OOP. It was just as important in Structured Programming.
- There are two aspects to encapsulation as we use it in OOP.
  - One is selecting a group of things (data and/or operations) and putting them together to create a module.
  - The other is making some of the things accessible from outside the module, and making the rest inaccessible. (Information Hiding)
- To the rest of the program, the module is its public (accessible from outside) parts.
  - The public part of a module is sometimes called the *public interface*, or just the *interface*.
  - The private parts are of interest only to the module itself.
  - To the rest of the program the private internals of a module don't exist. It's a black box.
  - If the private parts change, that change is constrained to the module and does not propagate through the rest of the program so long as the public parts are unchanged.
- The data is usually private, and accessible only through public methods.
  - This means that the data can only be manipulated by code that understands it, its rules and representation, and that can be trusted to maintain it properly.
  - It's like a secretary who keeps the information organized and gives you copies of anything you want, but you're not allowed to open up the file drawers yourself.

An object contains both data and the operations that can be performed upon the data. An object encapsulates everything that does not need to be seen from outside. Particularly, the object encapsulates the data representation and the implementation (code) of the operations. These things may change without affecting the client code as long as the interface of the object does not change. An object is a black box.

In a sense, this is enforced abstraction. The code that uses an object (client code) not only doesn't make use of the object's low-level details, it's not allowed to.

One of the important ideas is that you don't have to know how an object works or even (in many cases) what it does, in order to use it. You just have to trust that if you pass it a message it is qualified to handle, it will do the right thing. The code that sends the message and the programmer who wrote that code do not have to know what the receiving object will do, only that it will be the right thing. Of course, along with this principle, comes the responsibility to make it true. The objects must do the right thing (in accordance with their current state and the state of the rest of the system) in response to every message.

ABSTRACTION

- *Abstraction* is ignoring those aspects of something that do not contribute to your task in order to focus on those aspects that do.
  - This is particularly useful in software development, since there is much more going on than we can understand at one time.
- Objects allow programmers to realize the benefits of abstraction.
  - The details of data representation and the details of the methods are hidden from the programmers who use the object in their code (client programmers).
  - All that the client programmer needs to see, or can see, are the operations that can be performed on the object and how to use them. Not how the operations work, only what they do.

For example, when we add two integers in a program we ignore what's going on at the level of the memory and the CPU registers.

Are simple integers objects? Do you happen to know whether integers are stored in memory on your machine with the most or least significant byte first? And what registers are involved in adding two integers?

OBJECTS

- An *object* is a kind of module, used in models and programs, that contains both data (often called *attributes* or *fields*) and behavior (*methods*).

- An object can send messages to other objects invoking their methods, and respond with appropriate behavior to similar messages from other objects.

- Methods are invoked in response to messages.

  - Physically, a message calls a method on the object that receives the message.

- For example, a **Date** object might have attributes called **day**, **month**, and **year**.

- The **Date** object might have methods to read and write the attributes.

  - These might be called **getMonth()**, **setMonth()**, **getDay()**, etc.; a customary way of naming what we call *accessor functions*.

- The **Date** object might have methods to support other operations too.

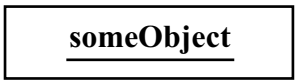
  - **addDays(int d)** to add a number of days to the object's data.

  - **addMonths(int m)**, **subtractDays(int d)**, etc.

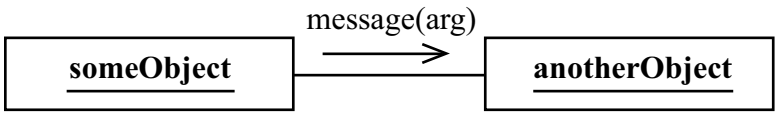
- Each object should have one primary responsibility that defines it, and may have more that are subsidiary to that one.

  - Even if you discover that an object you're designing has several different responsibilities, you should be able to think of a single one that includes them all.

  - Otherwise you may have a cohesion problem. Perhaps it should be several objects instead of one.



This is how we represent an object in UML: a box with an underlined name in it.



Here's how some object sends a message to another object in UML. Since a message is secretly a function call, it can have arguments like any other function call. The message can also return a value. This can be shown on the diagram like this:

```
localVar := getSomething(arg)
```

where **localVar** is a variable in **someObject**. Very often we don't want to be that detailed and so we omit the return. It's safe to assume that a function called **getSomething** gets something and that the calling code does something with it.

Objects should have high cohesion. They should represent a single thing or concept or design decision. They should have a single responsibility, or a set of closely related responsibilities.

Objects should have low coupling. They should not expose their inner workings, nor require that code that uses them understand more about them than their external interface.

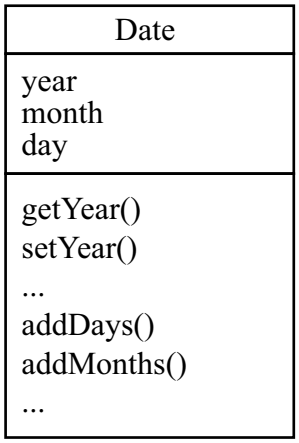
CLASSES

- A *class* is like a template for creating objects.
  - The class is the definition, description, or blueprint of the object.
  - The class tells what attributes and methods each instance of that class will have.
  - The class also describes the public interface through which other objects can access objects of this class.
  - The class is where the code for the methods is held.
- Every object is an instance of some class.
  - Just as a variable in a program might be an instance of a simple integer.
- Then, in the program, you can create as many objects or instances of a particular class as you need, and they will each have everything declared in the class.
  - Just as you can create all the integers you need in your program, and each one has the data and operations of an integer.

Class names are capitalized. This is a custom, not required by the compiler, but it is a very widespread custom that helps make programs more readable. If you don't adhere to it, other programmers may throw phone books over the walls of your cube.

Here is the UML symbol for the **Date** class.

Here's the **Date** class with attributes and operations.

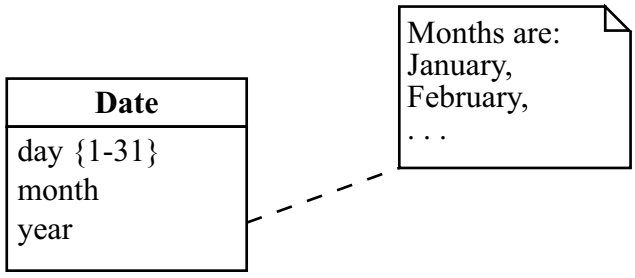


Attributes go in the second box, operations or methods in the third.

ATTRIBUTES

- An *attribute* is a data item that belongs to an object; it is a member of an object.
  - The attributes of a book object could include the title, author, publisher, and ISBN.
  - Attributes are also known as *fields*.
- The values of an object's attributes determine the object's state.
- Attributes are usually private, and thus not accessible from outside the object.
  - Outside code must use the public methods of the object to access its attributes.
- The datatypes of the attributes are often omitted in the early stages.
  - In general, it's often useful to put off details as long as you reasonably can in order to concentrate on the concepts.
  - If you make such decisions late you'll know more and you're more likely to get them right the first time. It's easier to defer them than change them.

The attributes go in the second box. We usually assume in the early stages that all data is hidden in the object and cannot be accessed from outside.



There are often rules (constraints) to govern the values of the attributes. There are only twelve months. Each has a maximum number of days. There are leap years. All these rules are written into the code that is part of the object so that the data is always maintained in a legal state and the outside code never has to worry about it. Such rules can be represented in a UML model as text in a box with its corner turned down or as a constraint in curly braces. The box above specifies the legal values for the attribute month. The constraint in the curly braces specifies the legal values for day. You may also specify constraints and relationships between attributes in a separate text file associated with the object. The constraints on a Date are too complex to put in the above diagram. You'd create a separate text document. An outside function can add a day to a date and rest assured that the Date object will handle wrapping around to the next month or next year as necessary.

The standard UML syntax for the datatype of an attribute is:

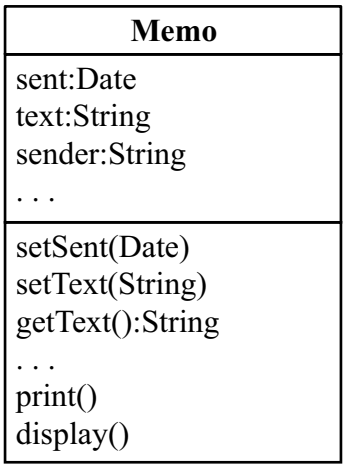
```
year : Integer
```

but many people use programming language syntax instead. Thus a Java programmer might write:

```
int year;
```

COMPOSITE CLASSES

- A class can contain an object as an attribute.
  - For example, a **Memo** class might have a date associated with it, which we might want to represent with a **Date** class.



  - We can represent that by putting a **Date** attribute inside the **Memo** class, using the class name as the datatype.

METHODS

- A *method* is a function that belongs to an object.
  - These are sometimes called *member functions* because they are members of the class.
- A method is invoked by sending a message to a particular object.
  - Thus a method is (almost) always invoked on a particular object. (We will see the exception later.)
  - The method can operate on that object's attributes.
  - Java programmers usually say "calling a method on an object" instead of "sending a message."

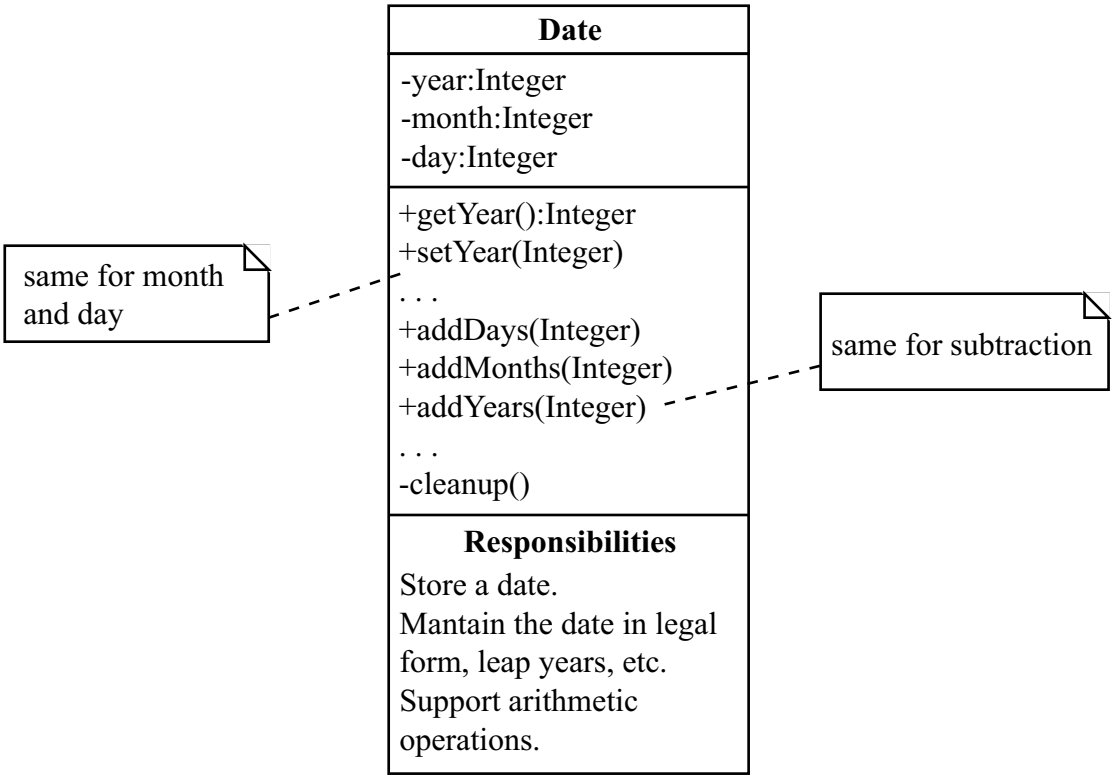
Methods go in the third box. In the early stages, when we first start thinking about an object, we may omit the arguments and return types from most or all of the operations. We put them in as we need them.

We usually assume that **getYear()**, for example, will return the value of the **year** variable. This is a little presumptuous, because we are supposed to think of and define the interface without reference to the internal representation. So we will think of **getYear()** as returning the year part of the date and its type will probably be integer although that's not important yet. By the same reasoning, **setYear()** will take an integer argument and set the year part of the date to that value.

You can add another box on the bottom for anything else you want to add, such as responsibilities or revision history, so long as you label it. Remember not to overcomplicate.

VISIBILITY

- The members of an object, its attributes and methods, may be visible and accessible from outside the object or not.
  - We call accessibility from outside *public* and inaccessibility *private*.
  - In UML, public gets a plus sign at the left, private gets a minus.
  - Constants are often made public.
- Attributes are almost always private.
  - Occasionally you may have a constant attribute that's public.
  - It's rarely reasonable to have a public variable.
  - Public variables break the encapsulation by allowing outside code to access the internals of the object.
    - This increases coupling.
    - If you change the attribute, the way it's represented, or anything about it, you risk breaking some unknown client code somewhere that uses it.
- Methods may be public or private.
  - If you have a method that is called by other methods in your object and you do not intend for client code to use it, make it private.
  - The **cleanup()** method in the **Date** class is called by other methods to make sure the invariants are satisfied (**months <= 12**, etc.) but we don't want to expose it as part of the interface to the object.
  - Public methods must be maintained forever because they are part of the object's published interface.



Usually, we start off assuming that all data is private and all operations or methods are public. We will find exceptions later, usually during design. In objects used in modeling the business (domain models) data is often public. Perhaps the engineering department secretary keeps the old blueprint files, but if you need a print after hours you can get it yourself. When you are finished with it, he'd rather that you left it on his desk instead of putting it back, because he knows how you file. This approximates read-only access.

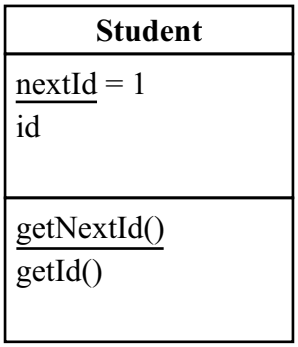
The **cleanup()** method contains the code that ensures that the values of the attributes always remain invariant, i.e. in their legal ranges. The other functions that change values of attributes all call **cleanup()** to roll over months and years and keep everything organized. It's private because we don't want code from outside the object using it.

CLASS SCOPE

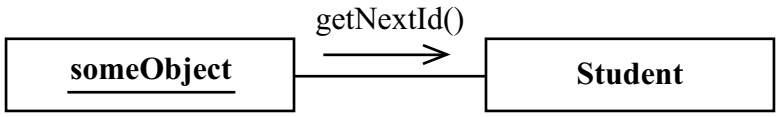
- An attribute of a class may have *class scope*.
  - This means there is one copy of the attribute regardless of how many instances of the class are created.
  - The class scope attribute exists even if there are no instances of the class.
  - All the instances of the class have access to this single copy of the class scope attribute.
  - The UML syntax for class scope is to underline the attribute name.
- A method may also have class scope.
  - A class scope method may be called on the class in a situation where there are no instances of the class, or the calling code does not have access to one.
  - Such a method may access class scope data but not ordinary (object scope) attributes. Since it may be called without an object, how would it know which object's attribute to use?
- Constants often have class scope.

Suppose we have a **Student** class and each Student is to have a unique **ID** number. These numbers shall begin with **1** and run sequentially as we register students in our university. The issue of managing this, providing the next number as we create each new **Student** object, is not trivial, especially if we may create **Student** objects in several places in our code. We'd like to handle it within the **Student** class if we could, and it turns out that we can.

We create a class scope attribute called **nextId** and initialize it to **1** when the program loads. (The means of doing this varies with each language.) Then each time we create a new instance of **Student**, the constructor writes the value of **nextId** into the new object's id attribute, and increments **nextId**.



If we need to read the value of **nextId** from some code that does not have access to a **Student** object, we can write a class scope method (**getNext-Id()**) which can be called on the class instead of on an object of the class. Here's the UML:



Note that **someObject** passes the message to the **Student** class, not to an instance of it. Also note that we do not explicitly show the return value from the message. We all know it's there. We'd show it if there was any confusion.

LABS

❶ Create a class diagram to depict a **Color**.

❷ Create a class diagram to depict a **Car**.

❸ Create a class diagram to depict a **BankAccount**.

❹ Create a class diagram to depict a class of your choice.

CHAPTER 4 - CLASSES AND THEIR RELATIONSHIPS

OBJECTIVES

- Model classes and static relationships between them, using UML.
- Add multiplicity and role names to your class diagrams.

CLASS MODELS

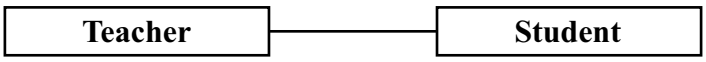
- The class model is the central model in OOA&D.
  - The most common class model is the UML *class diagram*.
- A *class model* is a static model of one or more classes and (optionally) the relationships among them at a particular level of abstraction.
  - It can show most of the characteristics of objects that we talked about in the object chapters.
  - A class model (or a set of them at different levels) can represent a complete object-oriented program.
- A model of a single class can show its attributes with or without datatypes, its methods with or without arguments and return types.
- A class model can show several classes and the relationships among them.
  - In this case, diagrams of the individual classes may omit some or all of the attributes and methods.
  - You should show only what contributes to the purpose of the model.
- Because a class model is so flexible and can represent so many different details from different levels of abstraction, it's especially important to choose only those parts that contribute to the model's purpose.

This chapter begins our study of the UML models. We'll cover the most useful and important parts of UML models, but we won't cover every detail.

UML Distilled by Martin Fowler is an excellent introduction to the UML. It is a short, very well written book. The third edition covers UML 2.0.

ASSOCIATIONS

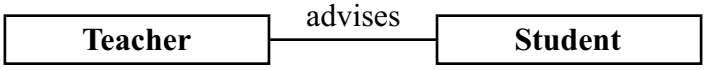
- *Association* is a generic relationship between classes.
  - If two classes are obviously related, but one is not part of the other and one is not a kind of the other, then their relationship is probably an association.
  - For example, consider **Student** and **Teacher**, the association that defines a school.



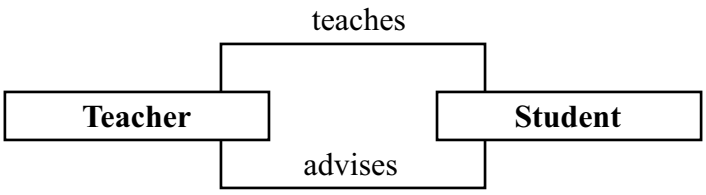
- Here we say there is an association between a **Teacher** and a **Student**.
  - There is a relationship between them, but neither is a part of the other and neither is a kind of the other.
  - Note also that the lifetimes of the teacher object and the student object may not be the same.

- The nature of this relationship is implied by the class names.
  - **Teacher** teaches **Student**.

- Where the relationship is not obvious from the class names, put a word or phrase on the line to explain it.
  - A **Student** may have an advisor who is a **Teacher**.

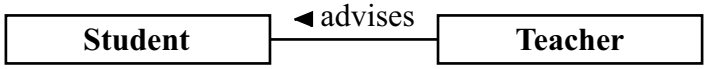


- There may be more than one association between two classes.



If you are unsure what type a particular relationship is, it's convenient to make it an association. You can change it later when you know more. Choose a verb for the name of the association that reads well, as in "Teacher advises student".

The name of an association normally reads left left-to-right or top-to-bottom. If the rest of the diagram forces you into a configuration where the association reads backward, you can show that with a small filled triangle.



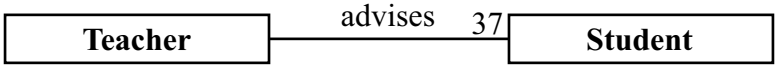
This still reads "Teacher advises Student." This is preferable to making the association passive. Passive constructions. such as "Student is advised by Teacher." are to be avoided, as in any writing. Lay out your diagram so that the names of the relationships read correctly (left-to-right, top-to-bottom) as much as possible. It's one of the little things that make models elegant and easy to understand.

We call them class models and the modules are classes, but the relationships we show on class models are between instances of the classes rather than the classes themselves. The class of **Teacher**s does not teach the class of **Student**s; a teacher object teaches a student object.

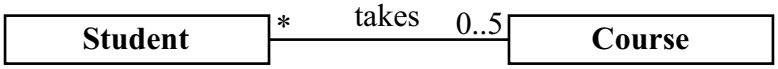
Class models, being static, cannot easily model interactions between classes and they also cannot model changing relationships. For example, an order clerk creates an order and populates it with line items. The order clerk has an association with the order by virtue of creating it, and needs the association so he can add the items. After the order is complete the order clerk passes the order on to the warehouse and never sees it again. The association between the clerk and the order is broken, but there is a new one between the order and the warehouse. This is not easily shown in a class diagram. Interaction diagrams can model this better.

MULTIPLICITY

- An association can involve more than one object on each side.
  - Suppose each teacher advises exactly 37 students.



- Show how many objects of a class can participate in the association by putting the number on the association line next to the class.
  - The default is exactly 1, thus the above diagram says each teacher advises exactly 37 students, and each student is advised by exactly 1 teacher.
- In the diagram below we say each student takes from zero to five courses while each course is taken by zero to many students.
  - The asterisk means any positive number, including zero.
  - 0..*n* or simply *n* mean the same as *, although it's not standard notation.



- These numbers pertain to a particular moment, not the lifetime of the system.
  - A student may have several advisors before graduating, but at a particular time will have only one.

The two multiplicities are independent. The number at one end of the line refers to the number of objects of that type that can be associated with a single object of the type at the other end.

Symbol	Meaning
1	exactly one
*	any (positive) number, zero to many
1..*	any positive number except zero, one to many
0..5	zero, one, two, three, four, or five
0,5	zero or five (not in UML 2.0)
	exactly one (default)

Attributes that are collections use this same notation to show numbers. For example, if a class contains a collection of one to many students, or one to many pointers or references to students it would be shown thus:

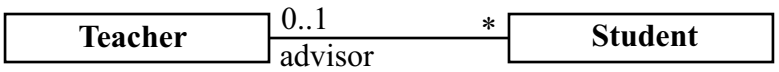
Students[1..*]

Note:
The square brackets mean any kind of collection, not necessarily an array.

ROLES

- There is another way to show that the **Teacher** is associated with the **Student** in a role other than **Teacher**.

  - Instead of naming the association (**advises**) as we did previously, we can name the role.



- Here we say there is an association between a **Teacher** and a **Student** in which the **Teacher**'s role is **advisor** (rather than the default role of **Teacher**).

  - It has exactly the same meaning as the diagram in which the word **advises** appears on the middle of the line. It would be redundant to use both.

  - Likewise, we could document the relationship by putting the word **advisee** on the line next to the **Student**, but **advisor** is a simpler and more common word. What's more, it's active voice, rather than passive.

LABS

❶ We are modeling the domain. Here's what we've discovered so far about the paper system we are going to replace:

a) Customers place orders.
b) An order has an order number, a date, a customer, and some line items.
c) A line item specifies a product and the number ordered.
d) Customers have credit ratings. If their credit is good we will ship and bill them later. If their credit is bad they must prepay.
e) Customers get various discounts, depending on what our salesperson has negotiated.

Create a class diagram showing these ideas and anything else that seems appropriate.

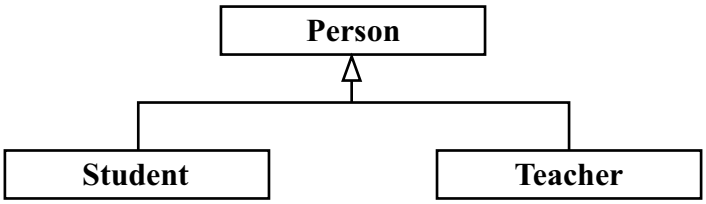
CHAPTER 5 - INHERITANCE

OBJECTIVES

- Use inheritance to represent a relationship between objects in which one object is a special kind of another object.
- Design with interfaces and abstract classes.

INHERITANCE

- *Inheritance* allows you to create a new class that contains all the attributes and methods of an existing class and then add to them.
  - The class you inherit from is often called the *superclass*, and the inherited class is the *subclass*.
  - The subclass may add new attributes, but it always has all the attributes of the superclass.
  - The subclass may add new methods, but it always has all the methods of the superclass.
  - The subclass may override one or more of the methods of the superclass by providing different implementations (code) for them.
- A subclass also inherits the responsibilities of the superclass.
- Inheritance is first a logical relationship.
  - The concept "subclass is a kind of superclass" should make sense.
  - A student is a kind of person.
  - A teacher is a kind of person.
  - Inheritance is sometimes called *specialization*. A teacher is a specialization of a person.
- Inheritance helps us avoid representing the same thing twice by allowing us to put attributes and methods that are common to several classes (subclasses) in a single common superclass.
- Inheritance helps us to reuse code when we create a new class by allowing us to inherit methods and attributes from an existing class that already has part of what we need in the new class.

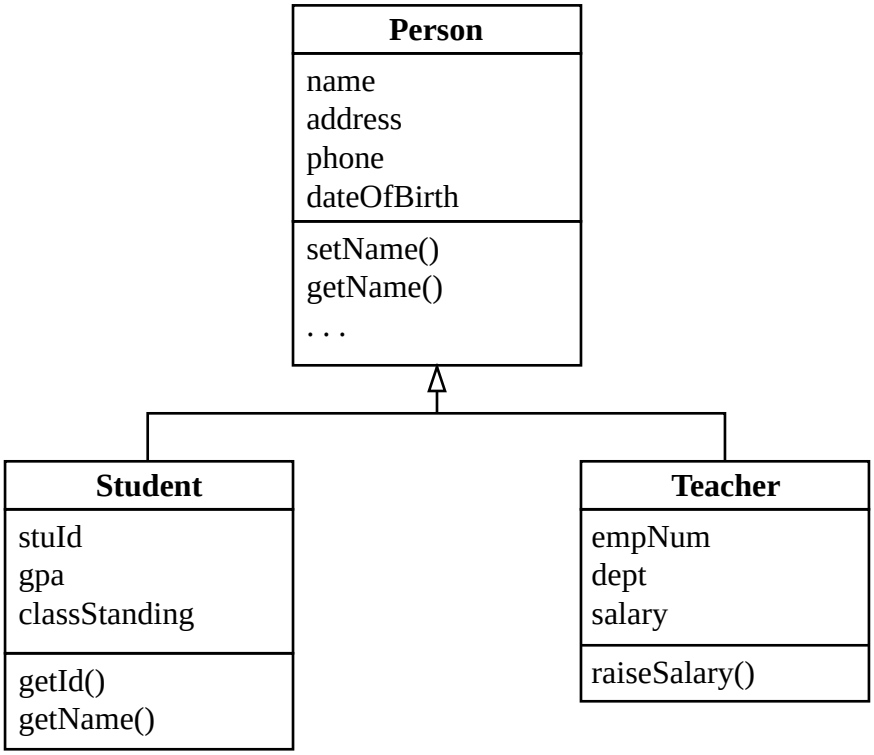


Other words for them:

Superclass — subclass (This is traditional OO terminology)
Base class — derived class (This is C++ terminology)
Parent class — child class
Ancestor class — descendent class

It would be nice to be able to see how this is going to work out right from the beginning of your analysis on a project, but you probably know that won't always happen. Sometimes you'll notice later in the development that you have several objects that share some logical and physical elements so you will factor out those elements and put them in a base class. Be on the lookout. Improving the structure as you go like this is called *refactoring*.

INHERITANCE EXAMPLE



- Objects of the **Student** class have names and addresses and so forth, which they inherit from the base class **Person**.

- **Student**s also have attributes of their own, not inherited, such as **GPA**.

- Likewise with operations: some are inherited, some are unique to **Student**.

- Let's imagine that in our school we always format the names of students lastname first, while other people's names have their firstnames first.
  - The **getName()** operation in **Person** will format the name with the firstname first.
  - The **getName()** operation in **Student** will override the **getName()** operation that **Student** inherits from **Person** and will format the name lastname first.
  - **Teacher** simply inherits **getName()** from **Person**.

- Thus, when the program calls **getName()** on a **Person** or a **Teacher** the name is formatted firstname first, but when the program calls **getName()** on a **Student** it executes the **Student getName()** and formats it lastname first.

Sometimes it's useful to think of the subclass as being just like its superclass, with a few changes and additions. Other times it's more useful to think of the subclass as containing a copy of the superclass inside. Sometimes it acts one way, sometimes the other.

When overriding a method from an ancestor class you must follow the Law of Least Astonishment and ensure that your method logically does exactly what the method you are overriding does. This is a crucial point.

In the example the **getName()** methods all do the same thing (get the name and format it as a string), but they do it appropriately for the different classes of object.

PROTECTED AND PACKAGE VISIBILITY

- The methods of a subclass cannot access private data or methods in its superclass, but must use the public methods of the superclass just as unrelated objects must.

- A third access specifier, *protected*, is indicated with a # at the left of the line.

  - Protected data or methods in the superclass are accessible by the methods of its subclasses and methods of other classes in the same package.

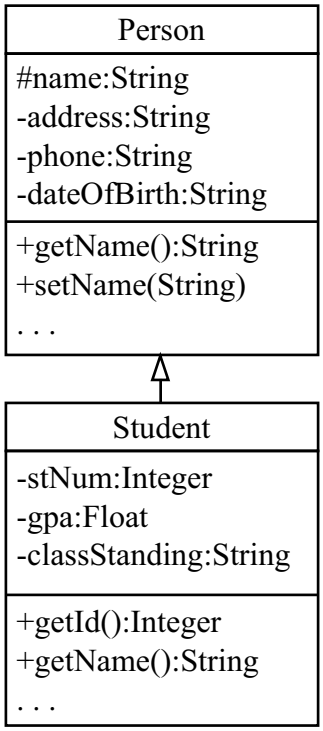
- Is the protected access mechanism necessary? Isn't it reasonable, for efficiency and convenience, for subclass objects to be able to access all superclass data directly?

  - It depends on how the classes will be used.
  - The protected attributes and methods are part of the published interface of the class when other classes inherit from it.
  - If other programmers will be inheriting from your class (which thus becomes a superclass), then you will not be able to change the protected attributes or methods without potentially breaking code in subclasses.

- It's better to make attributes private by default, and change specific attributes to protected only if you need to, and when you know that few or no others will be inheriting from your class.

- The last access class is *package*, indicated with a tilde (~).

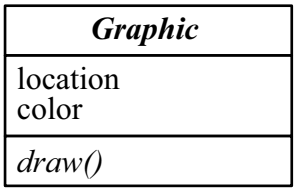
  - Package visibility means that it is accessible to any code in the same package, including code in packages contained within this one.



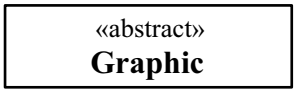
The **getName()** method inside the **Student** class has direct access to the name field inside the **Person** class.

ABSTRACT CLASSES

- An *abstract class* is one that cannot be instantiated; it exists for the sake of an inheritance hierarchy.
  - In UML, show that a class is abstract by italicizing its name.



  - You can also use a *stereotype*.



- Abstract classes often contain one or more method declarations, which have no code.
  - Note that **draw()** above is in italics to indicate that it has no implementation (code).
  - The **Graphic** class could not draw itself even if it had code. It doesn't know what shape it is.
- Abstract classes are intended to be used by subclasses that provide implementations for the code-less methods.
  - If a class inherits from an abstract base class, but does not provide code for all of the methods that have no code in the abstract class, the subclass will also be abstract.
  - Any class that has methods without code, or that inherits methods without code but does not provide their code, is abstract and cannot be instantiated.

It's impossible to create a modeling language (or any other kind of language) that will cover all possible cases. The *stereotype* is one of the ways we can extend the UML to communicate things that the creators did not anticipate. We can use it to create new specialized elements from existing general ones.

For example, consider the case on the facing page. We have a standard symbol for a class and we need to be able to represent a special kind of class, called an abstract class. We do this by putting the word **abstract** in guillemots, the small double angle brackets. If you can't find the «guillemots» on your keyboard, you can use double angle brackets for <<**stereotype**>>.

POLYMORPHISM

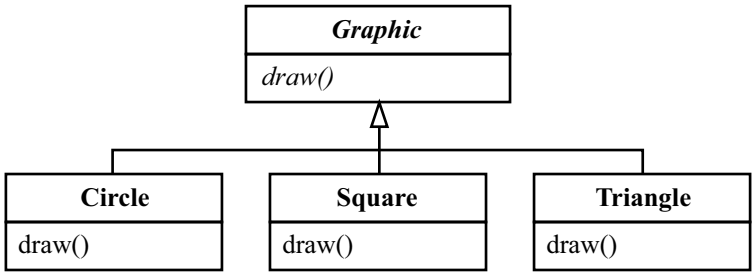
- *Polymorphism* allows a program to create objects of different (but related) types and manage them all in the same way, using the same code.
  - The program may store the objects in a single collection.
  - The program may send a particular message to one or more of them and they will each respond in a way that is appropriate for their class.
  - The code that manages the objects and sends messages to them need not know the types of the individual objects.
- Polymorphism is a powerful technique that can make your code much more readable, or much less.
  - The various methods in the subclasses that implement a single method declaration in the superclass may do physically different things, as appropriate for their classes, but must all do the same logical thing.
  - From the point of view of the client code, the code that uses them and manages them, all methods with the same name must do the same thing.
  - The logical operations are the same, even if the methods are different.
  - Remember the principle of least astonishment and don't surprise anyone.
- In strongly typed languages like C++ and Java polymorphism is implemented among classes that share a common ancestor.
- Polymorphism requires *dynamic binding* (also called *late* or *runtime binding*) as opposed to *static binding* (also called *early* or *compile-time binding*). Most of the older languages like C and Pascal use static binding.
  - In dynamic binding, the call to a method is not bound to the actual code until runtime.
  - Java normally binds at runtime, unless you specify that a method will not be used polymorphically, in which case it will be bound at compile time.

This isn't really a new concept. Multiplying two integers is logically the same as multiplying two floating point numbers, but, physically, at the machine level, the two operations are quite different. They have different representations in memory, and different register-level operations. But from the point of view of the higher-level code, all we see in both cases is multiplication. That's polymorphism.

These are the primitive numeric types that many programming languages offer. The point is that even old languages that do not officially have objects still have many of the characteristics of OO. We just never thought about it.

POLYMORPHISM EXAMPLE

- We are writing a drawing program. We create an abstract base class called **Graphic**. It has a **draw()** function but no code for it, since a graphic has no shape and you can't draw it. Then we create subclasses called **Square**, **Circle**, **Triangle**, etc. Each provides code for the **draw()** method to draw its own shape.



- Now we write the code for the rest of the program. As a user creates shapes and places them on the screen, we create objects of the proper types. We store the shapes in a list of **Graphic**s.

  - One of our rules is that anywhere the syntax calls for an object of a given class, an object of any of its subclasses may be substituted (the *Liskov Substitution Principle*). We know because of inheritance that any of the child classes will have all of the parent class's external interface, so we know it's safe.

- When the user hits the refresh button, the program iterates through the list of Graphics and calls **draw()** on each. This works because **Graphic** has a **draw()** method. What we get, of course, is two triangles, a circle, a square, three more circles, etc. Whatever the user created, and the program put on the list.

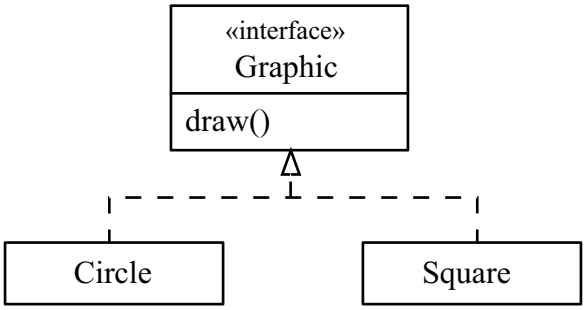
- Suppose we wish to later add pentagons. We write a **Pentagon** class as a subclass of **Graphic**, and add a pentagon button to the screen. The rest of the code remains the same. The user presses the refresh button and the program iterates through the list of **Graphic**s calling **draw()** and everything works. We've made a major change to the capability of the program with a minor change to the code.

As with inheritance, you will not always see all of these relationships clearly from the beginning. Sometimes the need for polymorphism shows up later, and you have to create it and the inheritance relationships that make it possible (in most languages) later.

INTERFACES

- An *interface* is like a class that has only a public interface and no private part.
  - An interface has no attributes (except possibly constants).
  - An interface has public operations (method declarations), but no code for them.
  - Interfaces are naturally abstract. They cannot be instantiated.

- Indicate an interface with a stereotype.



- We say **Circle** implements **Graphic** even though it looks like inheritance (note that the line is broken instead of solid).
  - This means **Circle** and **Square** inherit the method declarations of **Graphic**.
  - **Circle** and **Square** must provide code for **Graphic**'s draw operation or they become abstract and cannot be instantiated.

Note that polymorphism works with classes that implement the same interface just as it does with classes that inherit from the same ancestor. Of course, the only methods that can be used polymorphically are those from the interface.

LABS

❶ Create a class called **BankAccount** that has subclasses called **CheckingAccount** and **SavingsAccount**.

❷ Create a class called **Vehicle** that has subclasses called **Car** and **Bicycle**.

❸ Create a class called **Animal** with subclasses for various animals that you would find at a zoo.

❹ We are working toward the paperless office. As a start, we plan to store memos and orders on the computer. Your task is to make it possible for us to store the memos and orders associated with a particular customer in the same collection in such a way that they can be displayed or even printed if anyone needs a hard copy (polymorphism). Create and diagram the necessary classes.

INDEX

R

S

T

V

Skill Distillery

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
303-302-5234
www.SkillDistillery.com