The author of these slides, Zeegee Software Inc., has provided these slides for personal educational use only. Zeegee Software reserves all rights to the contents of this document.

Under no circumstances may the contents of this document, in part or whole, be presented in a public forum or excerpted for commercial use without the express permission of Zeegee Software.

Under no circumstances may anyone cause this electronic file to be altered or copied in a manner which omits, changes the text of, or obscures the readability of these Terms and Conditions.
INTRODUCTION

• What is/isn't object-orientedness?
• The "software crisis"
• Software reuse
• The evolution of good engineering practices
• The perils of non-OO development
What does "object-oriented" mean, anyway?

• Programs consist of entities, called objects, which correspond to concepts the program will manipulate:
  – A business application might have Employee and Department objects.

• Objects are created by specifying their structure and their properties in a class.
  – The Employee class is a (single) blueprint used to create (many and varying) Employee objects.

• Objects communicate by sending each other messages:
  – We might ask an Employee object "what's your name?".
What is object-orientedness?

• A **programming style** where concepts in your problem domain are mapped directly into your code: "object-oriented" means "concept-oriented."

• A **philosophy** of design and implementation (i.e., a way of thinking about your code) which can be employed even in non-OO programming domains.

• A **technology** to help you follow this philosophy: OO languages, OO databases, OO expert systems, etc.

• A **buzzword** used by people who want to sell you something.
What is object-orientedness not?

- It is not a solution to all your design and implementation problems.
- It will not allow non-programmers to "program like the pros."
- It does not allow you to side-step the design process and begin coding applications from day zero.
- It is not inherently better than non-OO for all tasks.
- It will not grow hair, lower your cholesterol, taste as fresh as homemade, soften hands while you do the dishes, and paint any car for $99.95… beware of sales pitches, no matter how flashy.
The software crisis: why we must change our ways

• Human society depending more and more on software.

• The *software crisis*: much existing software is...
  – bug-ridden, *because*...
  – too complex to understand, *and so*...
  – difficult to maintain, *and so*...
  – difficult to extend, *and so*...

• We reinvent the wheel over and over, leading to...
  – unnecessarily high development costs
  – new and more exciting bugs...
The goal: software reuse

• Programs tend to be "stick built" from the ground up:
  – Like making custom nails, screws, bricks, etc. for a house!

• Take our cue from the industrial revolution: assembly lines and interchangeable parts:
  – **Goal:** develop manageable, understandable, reusable software components that can be employed in a wide variety of applications, so that "new" code is specific to the problem at hand.

• Reuse is **not** the same as "cut and paste"; however...

• **Temptation** to "cut and paste" indicates useful code!
Benefits of software reuse

• Reduces coding/testing, thereby reducing delivery time and cost.

• Many reusers means many testers: well-tested code!

• Bugs found in reusable module can be reported to source: fix is made, new module is distributed, and now everyone benefits.

• Application programmers don't need to be experts in a wide variety of esoteric disciplines: easier to hire developers, and easier to keep them sane.

• High visibility of code can improve attitude of developer, and inspire more care and forethought, rather than just "task at hand" thinking.
Come on... how long does software last, anyway?

• Long enough for the Y2K bug to have been of some concern!
  – Much software written before 1990 did not have Y2K in mind.
  – Code from the 1980s is still in use!
  – People often "cut and paste" old code into new systems: the ghosts of programs long dead.
  – Developers are unwilling to tweak/upgrade components if they "seem to work fine as-is".

• Huge sums of money were spent to head off Y2K disaster. We'll never know if it was necessary, but we do know the price tag!
Modular programming

• Break large problems down into smaller, more manageable ones by breaking large programs into smaller pieces.

• Each piece can be coded and tested individually.

• Core of modularization is the subroutine (invented 1950s).

• But we still need to decide...
  – What tasks are made into subroutines?
  – What are the arguments? Return values? Side effects?

• A discipline was needed...
Structured programming

• Invented 1960s, as a means of good modularization.

• Relies on *functional decomposition*:
  
  – Top-down approach.
  
  – Break tasks (functionality) down from top level on down, to smaller and smaller components, until you get to actual subroutines.
Limitations of structured programming

• Can't always anticipate functions of evolving systems!
• The more successful a system is, the greater the chance it will be requested to handle a wider range of tasks than originally envisioned:
  – System gets "tweaked" to handle initial modifications.
  – Tweaks become "hacks" as desired functionality departs more and more from original decomposition.
  – System becomes more complex and unstable; further modifications require worse hacks or full rewrite.
  – Process stops when costs outweighs benefits.
Global/shared data: the break in the modularity

- At some point, many programs use *global data* which is shared between different subroutines.

- When any subroutine affects the global data, they can affect the other *subroutines* as well, in unexpected ways!
Modularization of data

• Often in structured programming, each subroutine is given its "own little world" of data:
  – Arguments
  – Local variables
  – Return value

```c
float hypotenuse (a, b)
    float a;
    float b;
    
    float c;
    c = sqrt((a * a) + (b * b));
    return c;
```

• One way structured programming helps is in modularizing data as well as functionality.

• But sometimes, we still have to share information!
The status quo: non-OO programming

- Programs consist of two separate entities: **data structures** and **functions**.
- Data structures: integer, string, array, list, stack, binary tree, hashtable, person, etc.
- Data structures are **accessed and modified directly**, or else by calling functions to perform the desired operations.

```c
IntStack s;
s.size = 0;
s.top = NULL;

istack_push(101, &s);
x = istack_pop(&s);
```
Problems with non-OO programming

• Functions for manipulating the same data structure can differ in naming convention, argument order, etc.: difficult to build/understand code:

```c
s.size = 0;
count = s_numelems(&s);
istack_push(101, &s);
x = pop_int_stack(&s);
print_thing(INTSTACK, &s);
```

• Ability/requirement to examine/modify parts of data structure results in programs which will fail if data structure changes.

• Can never be sure we have all the functions we need to build complex applications with data structure.
Problems with non-OO programming

• To implement something similar (but not identical) to an existing data structure, must often duplicate large amounts of code.

• Containers (arrays, lists, search trees, etc.) which can store different data types at the same time require lots of additional bookkeeping in the code:

```c
for (i = 0; i < size; i++) {
    thing = slot[i];
    if (thing->type == INT) {
        print_int(thing);
    }
    else if (thing->type == STRING) {
        print_string(thing);
    }
    else if (thing->type == CIRCLE) ...
    ...
}
```
How does OO help?

• **Message passing** paradigm provides clear, consistent syntax for accessing/manipulating objects.

• **Encapsulation** provides way of making data and "helper functions" inaccessible to prying eyes and sticky fingers.

• **Inheritance** allows new data structures to be defined in terms of existing ones, re-using existing (and tested) code.

• **Dynamic binding** means that data structures keep track of their types, so users of the data structures don't have to.

• All objects come bundled with the complete set of functions that are needed to work with them.
THE OO UNIVERSE

• Objects
• Classes
• Instance variables
• Instance methods and messages
• Pure OO environments
• Class variables/methods
• Constructors/destructors
Objects

• An **object** is a bundle of information that **models some high-level concept** in the problem domain.

• Generally a 1-to-1 correspondence between "real things" in the problem domain and objects in a running OO program.
  
  — This is one reason that OO programs can be very intuitive to work with: they're a kind of "**virtual reality**".

• The structure and behavior of an object is defined in the object's **class description**...

---

You can think of objects as just another kind of data type, like integers and strings. For example, a **boolean variable holds a very simple object** which might model the state of an on/off switch...
Objects

An object for a person

Lastname: Smith
Firstname: Jean
Age: 35
Weight: 120
Car: Ford

An object for their car

Make: Ford
Model: Escort
License: THX-1138
Color: white
Miles: 64000
Classes and instances

• A **class** is a blueprint for creating many similar objects.

• The created object is an **instance** of that class.

• Objects created from the same class will have the same basic **structure** and **functionality**.
  
  – All cars created from the same Ford Escort blueprints will look and work basically the same.

• **Many instances** can be created from a single class.
  
  – Just as many Ford Escorts can be created from the same set of Ford Escort blueprints.

*Think of objects as structs (C) or records (Pascal): there's a single type definition (the class), but many different structs/records (the instances) can be created from it.*
Classes and instances

Class: Car

A Car instance

Make: Ford
Model: Escort
License: TX118
Color: white
Miles: 64000

A Car instance

Make: Ford
Model: Prefect
License: HHG42
Color: black
Miles: 42000

Make: String
Model: String
License: String
Color: Color
Miles: Integer
Instance variables

• An **instance variable** (or **attribute**) of an object is a piece of information attached to an instance (object).
  
  – The name of a Person object, the model and year of a Car object, etc.

• The instance variables that an object has are **defined in the object's class**: an object can usually have many instance variables, of many different types.

• **Each object is given its own private space to hold its instance variables.** Assigning a new value to an instance variable of one object does not affect the instance variables of any other object.

*If you think of objects as structs (C) or records (Pascal), then the instance variables are the structure elements.*
Instance methods

• When we define objects, we usually have an idea of what we want to do with them...
  
  – I'm dealing with Person objects in an employee database... I want to be able to ask each Person object their name, weight, and age.
  
  – I'm dealing with Car objects in a driving simulation... I want to be able to start a Car, change its speed, turn its steering wheel, etc.

• An action that involves a single object as the "star player" is usually implemented as a special kind of function/subroutine attached to that object's class, called an instance method (or, more commonly, just a method).
Methods and messages

• A **message** is the *request* you send to an object in order to get it to do something: perform an action, return a value, etc.

```java
someone.setName("Picard");
```

• A **method** is the *piece of code* which is called to perform the activity requested by the message:

```java
Person::setName(String newname){
    self.name = newname;
}
```

• **Message passing** and **method invocation** usually mean the same thing: the act of sending a message to an object, and having that object do something.
What messages look like

A message generally has three parts:

– The receiver: the object receiving the message.
– The method name: what we want the object to execute.
– The parameters: any arguments that the message takes.

`someone.setName("Jean-Luc", "Picard");`
What methods look like

• Like any other function, **methods can take arguments.**

• Methods know **which object received the message** (they have to... after all, it's the star player!).

They might access the object through a special variable (C++'s this, Smalltalk's self):

Or through a special argument given to the method (Perl, Python):

Or just access its instance variables implicitly (C++, Smalltalk):

```cpp
Person::setName(String new){
    this->name = new;
}
```

```perl
sub setName {
    my ($self, $new) = @_;  
    $self->{Name} = $new;
}
```

```cpp
Person::setName(String new){
    name = new;
}
```
An OO bestiary

Classes

Real world

Software

class Car {
    String itsModel;
    int itsYear;
    Tire itsTires[4];
    ...
}

Instances

car

Instance variables

seat
steering wheel
tires

Methods

dashboard

myCar.start();
myCar.shiftGear(FIRST);
myCar.accelerate();
myCar.turn(LEFT);

myCar

myMom'sCar
myFriend'sCar
class Person {

    String name;
    int age;
    float weight;

    /* Change my weight: */
    setWeight(int newWeight) {
        weight = newWeight;
    }

    /* Return my weight, if I won't mind: */
    getWeight() {
        if (weight < 150)
            return weight;
        else
            return -1;
    }

    }
}
Using classes and objects

Person kirk("Kirk, James T.");
kirk.setWeight(190);
kirk.setAge(37);
wgt = kirk.getWeight();

Create instance of class Person
Send messages to object "kirk"
Representing classes and objects

Class

Object

Classname

inst-var-1
inst-var-2
inst-var-3

Classname

inst-var-1
inst-var-2
inst-var-3

Classname

inst-var-1
inst-var-2
inst-var-3

method-1(args)
method-2(args)
"Pure" OO environments

• In very "pure" OO environments, everything is an object!
  – Every class might be an instance of a special MetaClass class, which in turn might be an instance of itself!
  – Methods/functions might be instances of a special Code class.

• This can be taken to surprising extremes... for example, in OPAL (Smalltalk), there is no if-then statement: instead, you send an ifTrue:ifFalse message to a Boolean object:

```
method: getWeight
  (weight < 150.0)
  ifTrue: [^weight]  % okay
  ifFalse: [^-1]
  % no way!
```


If a class is just another kind of object, then it should have its own methods and respond to messages, right?

• Right! Because of this, we usually divide methods into:

  – *Class methods*, which are invoked when you send a message to a class.

  – *Instance methods*, which are invoked when you send a message to an instance of a "normal" class. We usually just call these methods.

• The most common class method is the *constructor*, which returns a new instance. It's often called "new":

```ruby
/* Class method: */
p = Person.new("Kirk");

/* Instance method: */
p.setName("Kirk");
```
Class methods, cont'd

Class methods generally do not operate on instances! They are intended for performing utility functions that are strongly associated with a class, but that don't involve any particular instance of that class!

• Other candidates for class methods:
  – Storing/retrieving objects of this class by some appropriate lookup key (e.g., the name).
  – Asking for information related to the class as a whole:

```cpp
/* Get the next available license plate. */
class method Car::nextLicensePlate {
    CurrentLicense += 1;
    return "ABC" + CurrentLicense;
}
```
Class variables

If a class is just another kind of object, then it should have its own instance variables too, right?

• Right! And these special instance variables are commonly called **class variables**.

• In many OO environments, the only places you can access a class variable are...
  
  – From inside a **class method** of that class
  – From inside an **instance method** of that class

• Class variables are kind of like global variables which are associated with a particular class.
Constructors

• As mentioned before, **constructors** are special functions (usually class or instance methods) used to **initialize or return a new object**.

• Most common name for a constructor is **new()**.

• Depending on the OO environment, a class **might have many constructors**, each of which builds an object a different way.

• Different constructors might have **different names** (Perl5, Smalltalk), or they might all have the **same name** but be distinguished by having different numbers/types of their arguments (C++).
Sample constructors

- In C++, constructors are special unnamed instance methods, invoked when you declare object variables:

  ```cpp
  // Create a Person from nothing:
  Person anonymous;

  // Create a Person from name, age, and weight:
  Person nsolo("Napoleon Solo", 35, 190.0);

  // Create a Person from another Person:
  Person clone(nsolo);
  ```

- In Perl5, constructors are class methods, and can be named whatever you like:

  ```perl
  # Create a Person from name, age, and weight:
  $nsolo = Person->new(
    Name=>"Napoleon Solo",
    Age=>35,
    Weight=>190.0);
  ```
Destructors

• Objects often end their lives by being explicitly deleted (C++), going out of scope (C++), or being garbage-collected (Java, Perl, Smalltalk).

• We sometimes want to get control of the object just before it vanishes, to clean up after it properly:
  – We may have opened files, which we want to close.
  – We may have allocated memory, which we want to free.

• The special instance method invoked just as an object is about to disappear is called the destructor.

• Classes generally have one destructor, which takes no arguments.
OO FEATURES/BENEFITS

• Encapsulation
• Polymorphism
• Subclasses and inheritance
• Class hierarchies and inheritance schemes
• Abstract/concrete classes
• Static/dynamic binding
Encapsulation

• There is a piece of wisdom which has made its way down through centuries of software development...

  *Teams work best when members of those teams know as little about each others' work as possible.*

• When one software module depends on the low-level implementation details of another module, unexplained bugs tend to appear as the system evolves.

• The "back room" metaphor.

• We need a way of not only *suggesting* that programmers stay out of the "back room", but *enforcing* it.
Encapsulation

- **Encapsulation** means that some or all of an object's internal structure is "hidden" from the outside world.

- Hidden information may only be accessed through the object's methods, called the object's *public interface*.

- Access to object is safe, controlled.

- Methods, like instance variables, may also be hidden to create private "helper functions".

- "public interface" methods (the hard candy shell)

- "private" internal structure/methods (the chocolate filling)
Objects as "goods" and "services"

What is your yearly income?

An object of class "Person"

Person::income()
{
    return (salary - expenses);
}

user of object

method (public interface)

structure (private portion)

method (public interface)
Encapsulation as maintainability

Person::income() {
    return (salary + (savings * intRate) - expenses -
            tax(salary, savings, deductions));
}

What is your yearly income?

user of object
method (public interface)
structure (private portion)

other methods

salary savings age
expenses
deductions
## Encapsulation for class builders and class users

<table>
<thead>
<tr>
<th>Class builder</th>
<th>Class user</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td><strong>Goal</strong></td>
</tr>
<tr>
<td>Create a reusable, maintainable class with an understandable interface.</td>
<td>Quickly get and use a class.</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td><strong>Approach</strong></td>
</tr>
<tr>
<td>Hide structure and implementation details from user.</td>
<td>Examine and use operations in class interface.</td>
</tr>
<tr>
<td><strong>Benefits</strong></td>
<td><strong>Benefits</strong></td>
</tr>
<tr>
<td>May change structure and algorithms without affecting user.</td>
<td>Understandable interface; can be used without fear of &quot;breaking&quot; object.</td>
</tr>
<tr>
<td><strong>Costs</strong></td>
<td><strong>Costs</strong></td>
</tr>
<tr>
<td>Reusable classes must be carefully planned.</td>
<td>Access to data through operations may be slower than direct access.</td>
</tr>
</tbody>
</table>
Polymorphism

• Literally, "one entity, many forms."

• Means the same name can be assigned to different functions/methods: the actual function triggered by the name is determined by the types of the arguments.

• Sometimes called overloading.

• Allows designers to use the most "natural" and understandable names for functions:

```c
/* Matrix M times vector v: */
y = M * x;

/* Matrix M times scalar: */
N = M * 2;

/* Vector x times scalar: */
y = 4 * x;
```

The multiplication operator is overloaded
With/without polymorphism

• **Without polymorphism**, we need a unique name for every function:

```c
/* Stuff with matrices, vectors, and scalars: */
mat_vec_mult(
    M,
    vec_times_scal(
        vec_cross(vec_minus(A, B), vec_plus(A, B)),
        -1)
);
```

• **With polymorphism**, we can use natural names and operators, shrinking code and increasing readability:

```c
/* Stuff with matrices, vectors, and scalars: */
M * (-1 * (A-B).cross(A+B))
```
More fun with polymorphism

Overloading '+' to perform string concatenation:

```
fullname = lastname + "," + firstname;
```

Overloading 'min' to compare different types:

```
firstInDictionary = min("astro", "zodiac");
lowerNumber = min(42, 99);
closerToOrigin = min(complex1, complex2);
```

Overloading 'print' to print different entities to different kinds of output:

```
aPerson.print(myFile);
aPerson.print(myScreen);
aPerson.print(myPrinter);
aPlace.print(myPrinter);
```
### Polymorphism for class builders and class users

<table>
<thead>
<tr>
<th></th>
<th>Class builder</th>
<th>Class user</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Create maintainable, reusable, class <strong>library</strong> with understandable interfaces.</td>
<td>Quickly get and use a class.</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>Use polymorphism to create <strong>similarity</strong> between class interfaces &amp; speed learning.</td>
<td>Examine and use operations in class interface.</td>
</tr>
<tr>
<td><strong>Benefits</strong></td>
<td>Interfaces to new classes are already partially designed.</td>
<td>Learn one class, and learn a bunch!</td>
</tr>
<tr>
<td><strong>Costs</strong></td>
<td>Must pick method names and arguments carefully. Must <em>not</em> reuse names if purposes differ!</td>
<td>Misnamed methods which are similar <em>but not identical</em> can be confusing.</td>
</tr>
</tbody>
</table>
Subclasses

• It is often desirable to create a new class which is a special case of an existing class, possibly with some small changes in structure or methods:

```java
class Person {
    String name;
    int age;
    float weight;

    void setWeight(wgt) {...}
    int getWeight() {...}
}
```

```java
class Doctor {
    String name;
    int age;
    float weight;
    School medSchool;
    String licenseNo;

    void setWeight(wgt) {...}
    int getWeight() {...}
    void examine(patient) {...}
}
```

• To re-use the existing code, make the new class a subclass of the existing one...
Subclasses and inheritance

• If class C is a subclass of class P, then C is a \textit{child class} of P, and P is a \textit{parent class} or \textit{superclass} of C.

• A child class automatically \textit{inherits} all the structure and functionality of its parent class. When defining a subclass, you will usually choose to:
  
  – \textbf{Add} new instance variables and methods.
  
  – \textbf{Override} some methods of the parent class, providing new methods.
  
  – \textbf{Exclude} some of the instance variables and methods of the parent class.
Adding structure/behavior in subclasses

When you specify instance variables/methods in a child class that are not present in the parent class, instances of the child class get the new structure and functionality in addition to what they would get from the parent alone:

**Classes**

class Person {
    int age;
    int getAge() {...}
}

class Doctor : Person {
    String licenseNo;
    void examine(patient) {...}
}

**Instances**

a Person:

<table>
<thead>
<tr>
<th>age</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAge()</td>
<td></td>
</tr>
</tbody>
</table>

a Doctor:

<table>
<thead>
<tr>
<th>age</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>licenseNo</td>
<td>AMA-1179150-6</td>
</tr>
<tr>
<td>getAge()</td>
<td></td>
</tr>
<tr>
<td>examine(patient)</td>
<td></td>
</tr>
</tbody>
</table>
Overriding behavior in subclasses

When you specify a method in a child class that already exists in the parent class, the child's method overrides the parent's method: instances of the child will execute the child's method.

```
class Person {
    String name;
    String getName() {
        return name;
    }
}

class Doctor : Person {
    String getName() {
        return "Dr. " + name;
    }
}
```
Overriding with default behavior

Sometimes it is desirable to override a parent's method, and yet also to *invoke the parent's method* to do some of the work:

```java
class Person {
    String first, mi, last;

    String getName() {
        return first + mi + last;
    }
}

class Doctor : Person {
    String getName() {
        return "Dr." +
            Person::getName();
    }
}
```

**Classes**

**Instances**

getName() returns "Leonard N. McCoy"

getName() returns "Dr. Leonard N. McCoy"
Exclusion in subclasses

If a child class is a very special case of its parent, it may not need all the instance variables in its parent. Some languages allow variables/methods to be excluded in a subclass.

```java
class Rectangle {
    int top;
    int left;
    int width;
    int height;
}
class Square : Rectangle {
    exclude width;
    exclude height;
    int side;
}
```

Classes

Instances

```
top 30
left 120
width 50
height 70
area()
```

```
top 30
left 120
side 60
area()
```
Class hierarchies

Usually, a class can have many child classes, each of which can have their own child classes. A "family tree" of all the classes in a program is called a **class hierarchy**...

The **base classes** of a given class $O$ are those classes from which $O$ ultimately inherits (parents, grandparents, etc.).

The **derived classes** of a given class $O$ are those classes which ultimately inherit from $O$ (children, grandchildren, etc.).
Inheritance schemes

One important way we characterize an OO language is by the kind of inheritance we are allowed to use...

• If classes can't be derived from parents, we have a *no-inheritance* language. Its class hierarchies will be flat.

• If each class can have at most one parent, we have a *single-inheritance* language. Its class hierarchies will resemble trees.

• If each class to can have more than one parent class, we have a *multiple-inheritance* language. Its class hierarchies will be graphs (DAGs).
A single-inheritance class hierarchy

more general

more specific

Object

Person

Doctor

Typist

Psychiatrist
A single-inheritance scheme

**Object**

**Person**
- name
- age
- weight
  - setWeight(wgt)
  - getWeight()

**Doctor**
- school
- license
  - getWeight()
  - examine(patient)

**Dentist**
- painless?
  - cleanTeeth(patient)

<table>
<thead>
<tr>
<th>name</th>
<th>McCoy, Leonard</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>52</td>
</tr>
<tr>
<td>weight</td>
<td>150</td>
</tr>
<tr>
<td>school</td>
<td>Johns Hopkins</td>
</tr>
<tr>
<td>license</td>
<td>AMA-1179150-6</td>
</tr>
</tbody>
</table>

An instance of "Doctor"
A multiple-inheritance scheme

Object

Person
    name
    age
    weight
    setWeight(wgt)
    getWeight()

Doctor
    school
    license
    getWeight()
    examine(patient)

NavyOfficer
    rank
    ship
    hoistTheMainsail()
    swabTheDeck()

NavyDoctor

name
age
weight
school
license
rank
ship

McCoy, Leonard
52
150
Johns Hopkins
AMA-1179150-6
CMO
U.S.S. Enterprise

An instance of "NavyDoctor"
## Advantages/disadvantages of using inheritance

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Code sharing.</td>
<td>• Requires an OO programming environment.</td>
</tr>
<tr>
<td>• Software reusability without disturbing working code.</td>
<td>• Can inherit attributes and operations that are not desired.</td>
</tr>
<tr>
<td>• Overloading helps develop standard class interfaces.</td>
<td>• Too many subclasses makes debugging difficult.</td>
</tr>
<tr>
<td>• Encourages design by extensions/refinements, rather than mass re-implementation.</td>
<td>• Performance may be degraded.</td>
</tr>
</tbody>
</table>
Abstract classes vs. concrete classes

- **Abstract classes** are classes which do not have instances of their own: they exist solely so that their child classes may inherit structure and/or functionality.

- **Concrete classes** are classes which may have instances. A concrete class may also have child classes.

```
Abstract class
Concrete class
```

```
Shape
  color
  pattern
  hasBorder

Polygon
  Regular Polygon
topLeft
  numSides
  center
  hasBorder

Irregular Polygon
  points

Round Rect
  radius

Circle
  center
  radius

Rect
  topLeft
  botRight

```

```
Shape
  color
  pattern
  hasBorder

Polygon
  Regular Polygon
topLeft
  numSides
  center
  hasBorder

Irregular Polygon
  points

Round Rect
  radius

Circle
  center
  radius

Rect
  topLeft
  botRight
```
In some OO languages, objects don't keep track of their types:

- An object's class (and, therefore, its behavior) is determined at compile-time by the class of the variable it's placed in (or referenced through).

In other OO languages, objects know their types:

- An object's class is determined at run-time. Not affected by the class of the variable the object it's placed in (or referenced through).
Behavior under static vs. dynamic binding

Recall our example where Doctor overrode the getName() method of Person. Now consider the following program:

```java
Doctor julian("Bashir");
Person someone = julian;
print(someone.getName());
```

If we get...

Bashir

...then the Doctor behaves like a Person after being stored in a variable of type Person: we've got static binding.

If we get...

Dr. Bashir

...then the Doctor behaves like a Doctor even after being stored in a variable of type Person: we've got dynamic binding.
Containers with static vs. dynamic binding

for (i = 0; i < size; i++) {
    Object *thing = slot[i];

    switch (*thing.type) {
    case INT:
        (*(IntObject *)thing).print();
        break;
    case STRING:
        (*(StringObject *)thing).print();
        break;
    case CIRCLE:
        (*(CircleObject *)thing).print();
        break;
    ...
    }
}

for (i = 0; i < size; i++) {
    slot[i].print()
}
# Static binding vs. dynamic binding

<table>
<thead>
<tr>
<th>Static binding</th>
<th>Dynamic binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of object fixed at compile time,</td>
<td>Type of object determined at run time, so objects</td>
</tr>
<tr>
<td>by type of variable containing it.</td>
<td>&quot;know what they are&quot;.</td>
</tr>
<tr>
<td>Often results in &quot;case&quot; and &quot;if&quot;</td>
<td>Eliminates complex &quot;case&quot; and &quot;if&quot; statements.</td>
</tr>
<tr>
<td>statements.</td>
<td></td>
</tr>
<tr>
<td>Heterogeneous collections difficult</td>
<td>Heterogeneous collections greatly simplified.</td>
</tr>
<tr>
<td>to implement.</td>
<td></td>
</tr>
<tr>
<td>Easier to debug.</td>
<td>Harder to debug.</td>
</tr>
<tr>
<td>No overhead at run-time.</td>
<td>Slower from run-time lookups.</td>
</tr>
</tbody>
</table>
What we've been leading up to: software reuse

The most important benefit of the OO approach is that by...

– Hiding implementation details (encapsulation)
– Reducing the number of different things you need to remember (polymorphism)
– Minimizing the amount of redundant code that needs to be built, tested, and maintained (subclassing)
– Helping existing code to not break just because someone has added a new data type (dynamic binding)

...it makes it far easier for you to reuse software components.
OO DATABASES

• What they are
• How they compare with Relational Databases
What is a database?

- A **database** is a repository for **persistent** information: information that exists even when no program is active, usually on disk.

- A good **database management system**...
  - Lets us **model** information in our problem domain accurately.
  - Allow **fast** insert, update, and query of the information.
  - Gives multiple users concurrent and reliable access to the information (**transaction management**).
  - Allows the information to be managed by **application programs** (not just through a user interface).
Relational databases

• Been around for decades.

• Information placed in *tables*:
  – Rows are individual data records.
  – Columns are the attributes of those records.

• Different tables are *joined* on common values of particular attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arthur</td>
<td>36</td>
<td>ZZ9ZA</td>
</tr>
<tr>
<td>Trillian</td>
<td>32</td>
<td>ZZ9ZA</td>
</tr>
<tr>
<td>Zaphod</td>
<td>34</td>
<td>HOG121</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Make</th>
<th>Color</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sirius</td>
<td>gold</td>
<td>HOG121</td>
</tr>
<tr>
<td>Nash</td>
<td>white</td>
<td>ZZ9ZA</td>
</tr>
</tbody>
</table>
Object-oriented databases

• An **OO database** (OODB) is like a running OO program, except that the objects are *persistent*: they are stored on disk rather than in memory.

• The objects in an OO database can send messages to each other, just as objects in OO programs can: message-sending can be used to perform queries, create and install new objects, send mail to users in a user database, etc.

• Ability to design and implement custom container classes allows developers to optimize database structure based on queries to be supported.
## Benefits of Relational DBs over OODBs

<table>
<thead>
<tr>
<th>OO</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Strong support in academic and commercial communities?
- Declarative vs. procedural query language?
- Underlying mathematical formalism for query optimization?
- Existing natural-language and GUI interface packages?
- Easy schema modification of large, populated databases?
## Benefits of OODBs over Relational DBs

<table>
<thead>
<tr>
<th>OO</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>?</td>
</tr>
<tr>
<td>✔️</td>
<td>?</td>
</tr>
<tr>
<td>✔️</td>
<td>?</td>
</tr>
<tr>
<td>✔️</td>
<td>?</td>
</tr>
<tr>
<td>✔️</td>
<td>?</td>
</tr>
</tbody>
</table>

- **OO**: Allows modeling of complex entities?
- **Relational**: Allows modeling of custom collections/indexes (such as quadtrees for fast spatial search)?
- **OO**: Does *not* require duplication of data?
- **Relational**: Scalable performance: will queries be fast for very large, complex databases?
- **OO**: Scalable storage: can database easily be distributed across many physical storage devices?
- **Relational**: Smooth interface to external environment (other databases, math libraries, email)?
When is OO good to use?

• Experience has shown that not all problem domains are perfectly suited for OO implementations. Things which can indicate OO as a **good** choice...

  – Need to manage complex information.
  
  – Need to model real-world concepts and processes (e.g., simulations).
  
  – Large number of developers to be coordinated.
  
  – Correctness, maintainability, and evolvability more important than blinding speed.
  
  – Rapid/evolutionary prototyping environment.
PROJECT: SIM

Many of you have played computer games where a realistic world is being modelled and maybe simulated: SimCity, Zork, Trinity, etc.

Ever wonder how people built those things?

Ever try to yourself?

Imagine that you want to create a swords-and-sorcery-style text adventure. Of course you're going to do an OO implementation. :-)

What classes will you need? What attributes and methods will they have? How will they interact? How "real" can/should the physics of your world be?