

Object Oriented Programming and Java

Note: See the sample code for examples of things mentioned here (these slide don't make sense without the code!).

Structured Programming

- Back in the "old days" we had Structured Programming:
 - data was separate from code.
 - programmer is responsible for organizing everything in to logical units of code/data.
 - no help from the compiler/language for enforcing modularity, ...

Hard to build large systems (not impossible, just hard).

OOP to the rescue

- Keep data *near* the relevant code.
- Provide a nice packaging mechanism for related code.
- Model the world as objects.
- objects can send "messages" to each other.

An Object

- Collection of:
 - Fields (object state, data members, instance variables, ..)
 - Methods (behaviors, ...)
- Each object has it's own memory for maintaining state (the fields).
- All objects of the same type share code.

Modern OOP Benefits

- Code re-use
 - programmer efficiency
- Encapsulation
 - code quality, ease of maintenance
- Inheritance
 - efficiency, extensibility.
- Polymorphism
 - power!

Code Re-Use

- nice packaging makes it easy to document/find appropriate code.
- everyone uses the same basic method of organizing code (object types).
- easy to re-use code instead of writing minor variations of the same code multiple times (inheritance).

Encapsulation

- Information Hiding.
- Don't need to know how some component is implemented to use it.
- Implementation can change without effecting any calling code.
- "protects us from ourselves"

Inheritance

- On the surface, inheritance is a code re-use issue.
 - we can extend code that is already written in a manageable manner.
- Inheritance is more, it supports polymorphism at the language level!

Inheritance

- Take an existing object type (collection of fields and methods) and extend it.
 - create a special version of the code without re-writing any of the existing code (or even explicitly calling it!).
 - End result is a more *specific* object type, called the sub-class / derived class / child class.
 - The original code is called the superclass / parent class / base class.

Inheritance Example

- **Employee**: name, email, phone
 - **FulltimeEmployee**: also has salary, office, benefits, ...
 - Manager: CompanyCar, can change salaries, rates contracts, offices, etc.
 - **Contractor**: HourlyRate, ContractDuration, ...
- *A manager* a special kind of **FullTimeEmployee**, which is a special kind of **Employee**.

Inheritance and Layers

- Layered systems allow specific functionality to be handled in one place (and only one place).
- Development can be incremental, develop each layer based on the others.
- Testing can be done on individual layers (test sub-functions directly instead of indirectly).

Polymorphism

- Create code that deals with general object types, without the need to know what specific type each object is.
- Generate a list of employee names:
 - all objects derived from Employee have a name field!
 - no need to treat managers differently from anyone else.

Method Polymorphism

- The real power comes with methods/behaviors.
- A better example:
 - shape object types used by a drawing program.
 - we want to be able to handle any kind of shape someone wants to code (in the future).
 - we want to be able to write code now that can deal with shape objects (without knowing what they are!).

Shapes

- Shape:
 - **color, layer** fields
 - **draw()** draw itself on the screen
 - **calcArea()** calculates it's own area.
 - **serialize()** generate a string that can be saved and later used to re-generate the object.

Kinds of Shapes

- **Rectangle**

Each could be a kind of shape (could be specializations of the shape class).

- **Triangle**

Each knows how to draw itself, etc.

- **Circle**

Could write code to have all shapes draw themselves, or save the whole collection to a file.

Java OOP

- Create new object type with `class` keyword.
- A class definition can contain:
 - variables (fields)
 - initialization code
 - methods

class definition

```
class classname {  
    field declarations  
    { initialization code }  
    Constructors  
    Methods  
}
```

Creating an Object

- Defining a class does not create an object of that class - this needs to happen explicitly:

```
classname varname = new classname();
```

- In general, an object must be created before any methods can be called.
 - the exceptions are *static* methods.

What does it mean to create an object?

- An object is a chunk of memory:
 - holds field values
 - holds an associated object type
- All objects of the same type share code
 - they all have same object type, but can have different field values.

Constructors

- Very similar to C++
- You can create multiple constructors, each must accept different parameters.
- If you don't write any constructor, the compiler will (in effect) write one for you:

```
classname() {}
```

- If you include *any* constructors in a class, the compiler will not create a default constructor!

Multiple Constructors

- One constructor can call another.
- You use "this", not the classname:

```
class Foo {  
    int i;  
    Foo() {  
        this(0);  
    }  
    Foo( int x ) {  
        i = x;  
    }  
}
```

A call to `this` must be the first statement in the constructor!



Destructors

- Nope!
- There is a `finalize()` method that is called when an object is destroyed.
 - you don't have control over when the object is destroyed (it might never be destroyed).
 - The JVM garbage collector takes care of destroying objects automatically (you have limited control over this process).

Class modifiers

- `public`: anyone can create an object of the defined class.
 - only one public class per file, must have same name as the file (this is how Java finds it!).
- default is non-public (if you don't specify "public").

Abstract Class modifier

- Abstract modifier means that the class can be used as a superclass only.
 - no objects of this class can be created.
- Used in inheritance hierarchies...(more on this later).

Field Modifiers

- Fields (data members) can be any primitive or reference type.
 - As always, declaring a field of a reference type does not create an object!
- Modifiers:
 - `public private protected static final`
 - there are a few others...

`public/private/protected` Fields

- `public`: any method (in any class) can access the field.
- `protected`: any method in the same *package* can access the field, or any derived class.
- `private`: only methods in the class can access the field.
- default is that only methods in the same package can access the field.

static fields

- Fields declared `static` are called *class fields (class variables)*.
 - others are called *instance fields*.
- There is only one copy of a static field, no matter how many objects are created.

`final` fields

- The keyword `final` means: once the value is set, it can never be changed.
- Typically used for constants.

```
static final int  BUFSIZE=100;  
final double  PI=3.14159;
```

Method modifiers

- **private/protected/public:**
 - same idea as with fields.
- **abstract:** no implementation given, must be supplied by subclass.
 - the class itself must also be declared **abstract**

Method modifiers

- **static**: the method is a *class method*, it doesn't depend on any instance fields or methods, and can be called without first creating an object.
- **final**: the method cannot be changed by a subclass (no alternative implementation can be provided by a subclass).

Method modifiers

- **native**: the method is written in some local code (C/C++) - the implementation is not provided in Java.
- **synchronized**: only one thread at a time can call the method.

Method Overloading

- You can overload methods:
 - same method name, different parameters.
 - you can't *just* change return type, the parameters need to be different.
- Method overloading is resolved at compile time.

```
int CounterValue() {  
    return counter;  
}  
  
double CounterValue() {  
    return (double) counter;  
}
```

Won't Work!

Inheritance vs. Composition

- When one object type depends on another, the relationship could be:
 - *is-a*
 - *has-a*
- Sometimes it's hard to define the relationship, but in general you use composition (aggregation) when the relationship is *has-a*

Composition

- One class has instance variables that refer to object of another class.
- Sometimes we have a collection of objects, the class just provides the glue.
 - establishes the relationship between objects.
- There is nothing special happening here (as far as the compiler is concerned).

Inheritance

- One object type is defined as being a special version of some other object type.
 - *a specialization.*
- The more general class is called:
 - base class, super class, parent class.
- The more specific class is called:
 - derived class, subclass, child class.

Inheritance

- A derived class object is an object of the base class.
 - is-a, not has-a.
 - all fields and methods are *inherited*.
- The derived class object also has some stuff that the base class does not provide (usually).

Java Inheritance

- Two kinds:
 - implementation: the code that defines methods.
 - interface: the method prototypes only.
- Other OOP languages often provide the same capabilities (but not as an explicit option).

Implementation Inheritance

- Derived class inherits the implementations of all methods from base class.
 - can replace some with alternatives.
 - new methods in derived class can access all non-private base class fields and methods.

This is similar to (simple) C++ inheritance.

Accessing superclass methods from derived class.

- Can use `super ()` to access all (non-private) superclass methods.
 - even those replaced with new versions in the derived class.
- Can use `super ()` to call base class constructor.

Single inheritance only (implementation inheritance).

- You can't *extend* more than one class!
 - the derived class can't have more than one base class.
- You can do multiple inheritance with *interface inheritance*.

Casting Objects

- A object of a derived class can be cast as an object of the base class.
 - this is much of the power!
- When a method is called, the selection of *which version of method* to run is totally dynamic.
 - overridden methods are dynamic.

Note: Selection of *overloaded* methods is done at compile time. There are some situations in which this can cause confusion.

The class `Object`

- Granddaddy of all Java classes.
- All methods defined in the class `Object` are available in every class.
- Any object can be cast as an `Object`.

Interfaces

- An interface is a definition of method prototypes and possibly some constants (static final fields).
- An interface does not include the implementation of any methods, it just defines a set of methods that could be implemented.

interface implementation

- A class can `implement` an interface, this means that it provides implementations for all the methods in the interface.
- Java classes can implement any number of interfaces (multiple interface inheritance).

Interfaces can be extended

- Creation (definition) of interfaces can be done using inheritance:
 - one interface can extend another.
- Sometimes interfaces are used just as labeling mechanisms:
 - Look in the Java API documentation for interfaces like **Cloneable**.