

# COMP2720: Automating Tools for New Media

## Objects and object-oriented programming

### Object-oriented programming

- First goal: Define and describe the objects of the world
  - Noun-oriented
  - Focus on the domain of the program
  - The object-oriented analyst asks herself: *"The program I'm trying to write relates to the real world in some way. What are the things in the real world that this program relates to?"*
- Example: Imagine you're building an O-O student information system
  - What are the objects?
  - Students, transcripts, classes, catalog, major-requirements, grades, rooms...

### History of objects: Where they came from

- Start of the story: late 60's and early 70's
  - Windows are made of glass, mice are undesirable rodents
- Good programming = procedural abstraction
  - That's basically what we've been doing — procedural-oriented programming
  - It's essentially Verb-oriented
    - We're defining "How to" swap backgrounds, increase red, decrease volume, etc.

### Alan Kay

*(regulator's warning: these accolades are included because A. Kay was the Guzdial's book referee)*

- University of Utah PhD student in 1966
  - Studied *Sketchpad*, the first object-oriented drawing program.
  - Studied *Simula*, a programming language designed to make simulations (e.g., to allow you to simulate a factory floor to see if the flow of materials worked well before you actually built it)
  - Plays jazz guitar (purportedly, as professional)
  - Has "traditional" (and wrong) views regarding Shakespear
- Saw *"objects"* as the future of computer science.
  - The way to build software better, more robustly, handle complexity better
  - Lead the team which created revolutionary Smalltalk software platform
  - Received A. Turing price in 2003

## Kay's Insights

- Think of the “computer” as a collection of networked computers.
  - Each one does its own thing and just communicates with others.
- All software is simulating the real world.
  - Therefore, it should be “noun-oriented” since the world is filled with nouns.
- Biology as model for objects
- “The best way to predict the future is invent it.”

## Birth of objects

- Objects as models of real world entities.
- Objects as Cells.
  - Independent, indivisible, interacting—in standard ways.
  - Scales well
    - Complexity: Distributed responsibility.
    - Robustness: Independent.
    - Supporting growth: Same mechanism everywhere.
    - Reuse: Provide services, just like in real world.

## Example to motivate objects: *SlideShow*

- Let's build a program to show a slide show.
  - It shows a picture.
  - Then plays a corresponding sound.
    - We'll use the `blockingPlay()` to make the execution wait until the sound is done.

## Slideshow

```
def playslideshow():
    pic = makePicture(getMediaPath("barbara.jpg"))
    snd = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("beach.jpg"))
    snd = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("santa.jpg"))
    snd = makeSound(getMediaPath("bassoon-g4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("jungle2.jpg"))
    snd = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(snd)
```

## What's wrong with this?

- From procedural abstraction:
  - We have duplicated code.
  - We should get rid of it.
- From *object-oriented* programming:
  - We have an object: A slide.

## Defining an object

- Objects *know* things.
  - Data that is internal to the object.
  - We often call those *instance variables*.
- Objects can *do* things.
  - Behavior that is internal to the object.
  - We call functions that are specific to an object *methods*.
    - But you knew that one already.
- We access both of these using *dot notation*.
  - `object.variable`
  - `object.method()`

## The slide object

- What does a *slide* know?
  - It has a *picture*.
  - It has a *sound*.
- What can a slide do?
  - Show *itself*.
    - Show its picture.
    - (Blocking) play its sound.

## Classes

- Objects are *instances of classes* in many object-oriented languages.
  - Including *Smalltalk*, *Java*, *JavaScript*, and *Python*.
- A class defines the data and behavior of an object.
  - A class defines what all instances of that class *know* and *can* do.

## We need to define a slide class

- Easy enough:

```
class slide:
    def someMethod(self):
        print "The slide class has one method."
```

- Methods are defined like functions, but indented within the class definition.
  - We'll explain `self` in just a few minutes.
- What comes next?
  - Some method for creating new slides.
  - Some method for playing slides.

## Creating new instances

- We are going to create new instances by calling the class name as if it were a function.
  - That will automatically create a new instance of the class.

## Creating a slide

- Let's create a slide and give it a picture and sound *instance variables*.

```
>>> slide1=slide()
>>> slide1.someMethod()
```

The slide class has one method.

```
>>> slide1.picture = makePicture(getMediaPath("barbara.jpg"))
>>> slide1.sound = makeSound(getMediaPath("bassoon-c4.wav"))
```

## Defining a show method

- To show a slide, we want to `show()` the picture and `blockingPlay()` the sound.
- We define the function as part of the class block.
  - So this is a `def` that gets indented.

## Defining the method show()

- Why self?

- When we say `object.method()`, Python finds the method in the object's class, then calls it with the object as an *input*.
- Python style is to call that *self*.
  - It's the object *itself*.

```
class slide:
    def show(self):
        show(self.picture)
        blockingPlay(self.sound)
```

## Now we can show our slide

```
>>> slide1.show()
```

- We execute the method using the same dot notation we've seen previously.
- Does just what you'd expect it to do.
  - Shows the picture.
  - Plays the sound.

## Making it simpler

- Can we get rid of those picture and sound assignments?
- What if we could call slide as if it were a real function, with inputs?
  - Then we could pass in the picture and sound filenames as *inputs*.
- We can do this, by defining what Java calls a *constructor*.
  - A method that builds your object for you.

## Making instances more flexibly

- To create new instances with inputs, we must define a function named `__init__()`
  - That's `underscore-underscore-i-n-i-t-underscore-underscore`.
  - It's the *predefined* name in Python for a method that initialises new objects.
- Our `__init__()` function will take three inputs:
  - `self`, because all methods take that.
  - And a picture and sound filename.
    - We'll create the pictures and sounds in the method.

## Our whole slide class

```
class slide:
    def __init__(self, pictureFile, soundFile):
        self.picture = makePicture(pictureFile)
        self.sound = makeSound(soundFile)

    def show(self):
        show(self.picture)
        blockingPlay(self.sound)
```

## Using map with slides

- Slides are now just *objects*, like any other kind of object in Python.
- They can be in lists, for example.
- Which means that we can use map.
- We need a function:

```
def showSlide(aslide):
    aslide.show()
```

## The playslideshow()

```
def playslideshow():
    slide1 = slide(getMediaPath("barbara.jpg"), getMediaPath("bassoon-c4.wav"))
    slide2 = slide(getMediaPath("beach.jpg"), getMediaPath("bassoon-e4.wav"))
    slide3 = slide(getMediaPath("santa.jpg"), getMediaPath("bassoon-g4.wav"))
    slide4 = slide(getMediaPath("jungle2.jpg"), getMediaPath("bassoon-c4.wav"))
    slide1.show()
    slide2.show()
    slide3.show()
    slide4.show()
```

## playslideshow with map

```
def playslideshow():
    slide1 = slide(getMediaPath("barbara.jpg"), getMediaPath("bassoon-c4.wav"))
    slide2 = slide(getMediaPath("beach.jpg"), getMediaPath("bassoon-e4.wav"))
    slide3 = slide(getMediaPath("santa.jpg"), getMediaPath("bassoon-g4.wav"))
    slide4 = slide(getMediaPath("jungle2.jpg"), getMediaPath("bassoon-c4.wav"))
    map(showSlide,[slide1,slide2,slide3,slide4])
```

## The value of objects

- Is this program easier to write?
  - It certainly has less replication of code.
  - It does combine the data and behavior of slides in one place.
    - If we want to change how slides work, we change them in the definition of slides.
    - We call that *encapsulation*: Combining data and behavior related to that data.
  - Being able to use other objects with our objects is powerful.
    - Being able to make lists of objects, to be able to use objects (like picture and sound) in our objects.
    - We call that *aggregation*: Combining objects, so that there are objects in other objects.

## We've been doing this already, of course.

- You've been using objects already, everywhere.
- Pictures, sounds, samples, colors — these are all objects.
- We've been doing aggregation.
  - We've worked with or talked about lists of pictures, sounds, pixels, and samples.
- The functions that we've been providing merely cover up the underlying objects.

## Using picture as an object

```
>>> pic=makePicture(getMediaPath("barbara.jpg"))
>>> pic.show()
```

## Slides and pictures both show()

- Did you notice that we can say `slide1.show()` and `pic.show()`?
- `show()` generally means, in both contexts, “show the object.”
- But what's really happening is different in each context!
  - Slides show pictures and play sounds.
  - Pictures just show themselves.

## Another powerful aspect of objects: *Polymorphism*

- When the same method *name* can be applied to more than one object, we call that method *polymorphic*.
  - From the Greek “many shaped”.
- A *polymorphic method* is very powerful for the programmer.
  - You don’t need to know exactly what method is being executed.
  - You don’t even need to know exactly what object it is that you’re telling to show()
  - You just know your goal: Show this object!

## Pictures and Colors have polymorphic methods, too

```
>>> pic=makePicture(getMediaPath("barbara.jpg"))
>>> pic.show()
>>> pixel = getPixel(pic,100,200)
>>> print pixel.getRed()
73
>>> color = pixel.getColor()
>>> print color.getRed()
73
```

## Uncovering the objects

- This is how the show() function is defined in JES:

```
def show(picture):
    if (not picture.__class__ == Picture):
        print "show(picture): Input is not a picture"
        raise ValueError
    picture.show()
```

- You can ignore the raise and if.
  - The key point is that the function is simply executing the method.

## We can get/set components at either level

- getRed, getBlue, getGreen, setRed, setBlue, setGreen
  - Are all defined for both colors and pixels.
- Why didn’t we define the functions to work with either?
  - It’s somewhat confusing to have a globally-available function take two kinds of things as input: Colors or pixels.
  - But it’s completely reasonable to have a method of the same name in more than one object.

## Sunset using methods

- Any of our older functions will work just fine with methods.

```
def makeSunset(picture):  
    for p in getPixels(picture):  
        p.setBlue(p.getBlue()*0.7)  
        p.setGreen(p.getGreen()*0.7)
```



## Backwards using methods

```
def backwards(filename):  
    source = makeSound(filename)  
    target = makeSound(filename)
```

To get the sample object, use:  
`sound.getSampleObjectAt(index)`

```
    sourceIndex = source.getLength()  
    for targetIndex in range(1,target.getLength()+1):  
        # The method is getSampleValue, not getSampleValueAt  
        sourceValue =source.getSampleValue(sourceIndex)  
        # The method is setSampleValue, not setSampleValueAt  
        target.setSampleValue(targetIndex,sourceValue)  
        sourceIndex = sourceIndex - 1
```

```
    return target
```

## Why objects?

- An important role for objects is to reduce the number of names that you have to remember.
  - `writeSoundTo()` and `writePictureTo()` vs. `sound.writeTo()` and `picture.writeTo()`
- They also make it easier to change data and behavior together.
  - Think about changing the name of an instance variable. What functions do you need to change? Odds are good that they're the ones right next to where you're changing the variable.
- Most significant power is in *aggregation*: Combining objects.

## Python objects vs. other objects

- One of the key ideas for objects was “not messing with the innards.”
- Not true in Python.
  - We can always get at instance variables of objects.
- It is true in other object-oriented languages.
  - In *Java* or *Smalltalk*, instance variables are only accessible through methods (`getPixel`) or through special declarations (“This variable is public!”)

## Inheritance

- We can declare one class to be *inherited* by another class.
- It provides instant *polymorphism*.
  - The child class immediately gets all the data and behavior of the parent class.
- The child can then add more than the parent class had.
  - This is called making the child a specialisation of the parent.
  - For example, a 3-D rectangle might know/do all that a rectangle does, plus some more:  
class rectangle3D(rectangle):

## Inheritance is a trade-off

- Inheritance is talked about a lot in the object-oriented world.
  - It does reduce even further duplication of code.
  - If you have two classes that will have many the same methods, then set up inheritance.
- But in actual practice, inheritance doesn't get used all that much, and can be confusing.

## When should you use objects?

- Define your own objects when you have:
  - Data in groups, like both pictures and sounds.
  - Behavior that you want to define over that group.
- Use existing objects:
  - Always—they're very powerful!
  - Unless you're not comfortable with dot notation and the idea of methods.
    - Then functions work just fine.

## What to do now

- Read chapter 14 in text book (covers lectures CSTopics-3 to CSTopics-6)
- Work on home work 2
- Work on portfolio
- Next week lecture(s) (30 October): *Course review and exam preparation*