Larger Programs

- As you become more skilled in writing programs, you will find that programs quickly increase to many, many lines of code.
- Games and other "real world" software applications can have thousands, even millions of lines of code.

Classes, Objects, & Methods

- Object-oriented programming uses classes, objects, and methods as basic programming components.
- These components help to organize a large program into small modules, design and think about an intricate program, and find and remove errors (bugs).

Classes

- In Alice, classes are predefined as 3D models

Objects

- An object is an instance of a class.
  - Class: Frog (Uppercase name)
  - Objects: frog, frog1, frog2, frog3 (lowercase names)

Objects

- An "object" is any thing that can be identified as unique from other things.
- How is an object unique?
  - has a name
  - has properties: width, height, color, location
  - can perform actions (methods):
    - associated actions it can perform
    - tasks it can carry out

Object Parts

- Objects may be composed of parts
3 Dimensions, 6 Directions
- A 3D object has
  - 3 dimensions
    - height, width, depth
  - 6 degrees of freedom (directions of movement)

Center of an object
- At the center of mass
- Where it stands on the ground
- Where it is held

Class
- Objects are categorized into classes
  - harry
  - ron
  - hermione
  - Each object is an instance of the class.
  - dog
    - fang
    - fluffy
    - snuffles
- All objects in a class have similar properties and generally can perform the same tasks.

We have also used...
- built-in (predefined) methods
  - Examples: move, turn to face, say
- World.my first method
  - Example:
    In the FirstEncounter world, we wrote program code where a robot was surprised by an alien.
    All the program code was written in this one method, see next slide...

Potential Problem
- The program code just seemed to grow and grow.
- If we continue to write programs this way the programs will become longer and more difficult to read and think about.
Solution

- A solution is to organize the instructions into smaller methods.
- A possible storyboard

**Demo: Starting a new method**

First, to associate the new method with the World
- select the World tile in the Object Tree
- select the methods tab in the details area
- click on the "create new method" button

**Demo: Starting a new method**

First, to associate the new method with the World
- select the World tile in the Object Tree
- select the methods tab in the details area
- click on the "create new method" button

**Solution**

- A solution is to organize the instructions into smaller methods.

**Top-Down Design: Modular Design**

- The next step is to break down each major task into simpler steps.
  - Example:
    
    ```
    surprise
    Do in order
    alien moves up
    alien says "Slithy toves?"
    robot's head turns around
    ```

**Stepwise Refinement**

- The process of breaking a problem down into large tasks and then breaking each task down into simpler steps is called stepwise refinement.
- Once the storyboard is completed, we write a method for each task.
Concepts illustrated in this example world:
- **surprise** is a **world-level method** because it is defined as a method for World and has instructions that involve more than one object (spiderRobot, alienOnWheels)
- The **surprise** method is executed by **calling** (invoking) the method.

**Why?**

- Why do we want to write our own methods?
  - saves time -- we can call the method again and again without reconstructing code
  - reduces code size – we call the method rather than writing the instructions again and again
  - allows us to "think at a higher level"
    - can think **surprise** instead of
    - the technical term for "think at a higher level" is **abstraction**

---

**Parameters**

Alice

---

**Storyboards**

- Each bug band member will perform a solo.

**A beetle band**

Our task is to create an animation for a bug band as an advertisement for their next concert.

---

**Demo**

Concepts illustrated
- To play a sound, a sound file (MP3 or WAV) must first be imported into Alice. (Alice is not a sound editor.)
- This code is only for georgeBeetle.
- Three more methods (one for each band member) will be needed!
A Better Solution

- Four versions of very similar code seems a bit tedious. The only things that change are the beetle and the music that plays.
- A better solution is to write a more flexible method.

Parameters

- Built-in methods provide flexibility by providing parameters such as distance and direction.
- Parameters allow you to pass in values (arguments).
  
  ![Parameters: distance, direction]
  
  Arguments: 0.5 meters, 0.5 seconds

Kinds of Parameters

- Alice provides several kinds of parameters that can be used in your own methods.

  ![Kinds of Parameters]

The storyboard

- In this example, we can write just one method and use parameters to specify:
  - which band member is to perform and
  - which music should be played.

  ![The storyboard]

Demo

- Ch04Lec2BeetleBand-v2
- Concepts illustrated
  - Enter name and select the type of each parameter
    - bandMember is an Object parameter
    - music is a Sound parameter
  - A parameter acts as a placeholder in the instruction
  - Arguments are passed to the parameter the call to the method

A Number parameter

- Add a Number parameter to specify the height the bandMember jumps up and down.

  ![A Number parameter]

  Note that the call to the method must now include an argument for the height.
Class-level Methods and Inheritance

Alice

Class-level Methods

Some actions are naturally associated with a specific class of objects.

Examples
- A person walking
- A wheel rolling

We can write our own methods to define an action for a specific class of objects -- a class-level method, instead of a world-level method we defined previously.

An example (building technique)

How can we create a `skate` method for ice skater objects?

We need to:
1. Tell Alice to associate the new method (the one we are about to write) with an ice skater, and
2. Write a new method to animate the ice skater skating.

Demo: The solution

First, to associate the animation with the ice skater
- select the iceSkater tile in the Object Tree
- select the methods tab in the details panel
- click on the `create new method` button

Storyboard for `skate`

The `slide` actions each require several motion instructions, so we will break down these two actions into smaller steps

Stepwise Refinement

Refinement of `slideLeft`
- Do in order
  1. Lift right leg and turn upper body forward
  2. Lower right leg and return body upright

Refinement of `slideRight`
- Do in order
  1. Lift left leg and turn upper body forward
  2. Lower left leg and return body upright
Concepts illustrated in this example world

- A method defined for a specific type of object defines an action for that object.
- A method can call other methods.

In this example, the `skate` method calls `slideRight` and `slideLeft`.

Writing methods to make an ice skater perform a skating motion is an intricate task.

We would like to have the `iceSkater` skate in other worlds without having to write the methods again.

The idea of being able to use previously written program code in another program is known as **reuse**.

Renaming `iceSkater` as `cleverSkater`.

1. Rename `iceSkater` as `cleverSkater`.
2. Save out as a new class. Alice saves the new class as `CleverSkater.a2c`.

The CleverSkater class **inherits** all the properties and methods from the original IceSkater class, and also

- has the newly defined methods (`skate`, `slideLeft`, `slideRight`)

In other programming languages, the concept of creating a new class based on a previously defined class is called **inheritance**.

An instance of the CleverSkater class can be added to a new world – use File|Import.

To avoid potential misuse of class-level methods, follow these guidelines (encapsulation):

- Avoid references to other objects
- Avoid calls to world-level methods
- Play a sound only if the sound has been imported and saved out as part of the new class

If these guidelines are not followed and an instance of the new class is added to another world, Alice will open an Error dialog box to tell you something is wrong.
Bad Example

What if there is no penguin in the new world where a cleverSkater object is imported?

Problem

Suppose you really want to write a class-level method where another object is involved?
For example, a method to make the skater skate around another object-- in this scene, the penguin.

Parameter

A solution is to write a class-level method with an object parameter that allows you to pass in the specific object.

```
cleverSkater.skateAround
Parameter: whichObject
Do in order
  Do together
    cleverSkater turn to face whichObject
    cleverSkater lift right leg
    cleverSkater move to whichObject
    cleverSkater turn around whichObject
```

Translation to Code

Most of the skateAround storyboard design is straightforward and easy to code.

One step, however, requires some thought:
```
cleverSkater move to whichObject --
what distance should the cleverSkater move?
```

Calling a built-in function

The instruction to move the skater to whichObject (penguin, in this example) would look like this:
```
cleverSkater move forward
cleverSkater distance to whichObject
```

Unfortunately, the skater will collide with the penguin because the distance between two objects is measured center-to-center.

Expression

To avoid a collision, use a math operator to create an expression that adjusts the distance.
Math operators in Alice:
```
+ addition  
- subtraction
* multiplication
/ division
```

Example:
```
cleverSkater move forward
=cleverSkater distance to whichObject
1
```
Demo

Ch04Lec3SkateAround

Concepts illustrated:
- A parameter acts as a placeholder for the object that will be passed in
- A call to the distance to function returns a number value
- A math expression can be created as part of an instruction