

CS112 –
Java
Programming

Spring 2024

Copyright 2023 Paul Haskell. All rights reserved.

Object Oriented Programming

So far...

We have learned about built-in data types and variables

We have learned some basic statements, including assignments and mathematical operations. And `println()` of course...

We saw the program control statements `while()` and `if()-else`

These are the core of Procedural Programming

Why isn't this enough?

Procedural Programming

Procedural Programming is not enough

- Not simple enough
- Not robust enough

...in cases where there can be hundreds or more types of user inputs (e.g. a GUI)

...in cases where dozens (or thousands) of people develop SW for the same project

...in cases where we must simplify the complexity of a sophisticated problem

Object Oriented Programming

Two key concepts

- **Class** – a user-defined data type that includes both data (variables, properties) and functions (methods, procedures)
- **Objects** – user-created variables whose type is some class. An object represents some real, important entity in our solution to some problem.

What's the big(gest) deal?

- **Creators** of a class can protect data from accidental corruption and can hide complexity...
- ...meaning **Users** of a class can use the class confidently, without fear of breaking something by accident
- ...and **Users** can think about their problem at a higher level. E.g. Math.log()

Encapsulation

WE WILL TALK MORE ABOUT THIS

Object Oriented Programming

We define a **class** to represent a student and **objects** represent actual people

- We create methods to add and drop courses, calculate GPA, print current schedule, add course grades, save changes to a central database
- We require an instructor password to add grades, and administrator password to add/drop courses, student password to see GPA

A **class** represents an image and **objects** are actual images

- We create methods to save an image as a JPEG file, increase/decrease brightness, resize the image, etc
- We do not allow the brightness to become negative, the size to become negative, etc

A **class** represents an item owned by the campus library. An **object** represents a particular item

- We store the item's title, publisher, publication date, length, etc
- We store whether the item is checked out and if so by whom and when due
- We store whether the item is physical (paper) or electronic
- We store its (physical or virtual) location

What programs might use this class? Course registration, signing up for dorm rooms, selling basketball tickets, figuring out who is graduating, how big classrooms are needed for classes, all sorts of things. These programs USE (but do not WRITE) class Student

Classes vs Object

↕ ↕
Data Types vs Variables – same thing

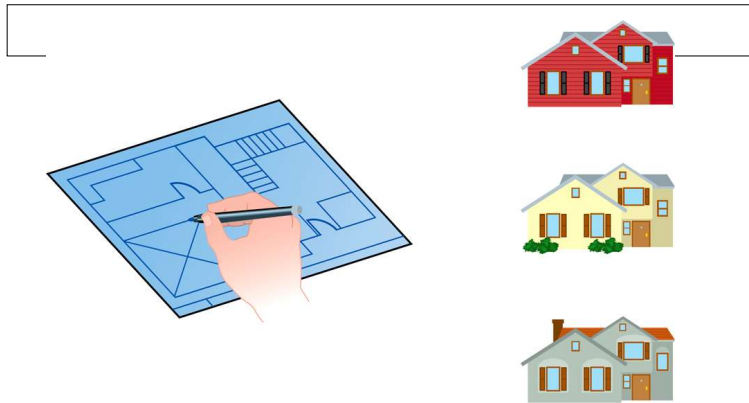
class Airline → Southwest, Delta, United, Virgin, American, KLM;

short → ageOfPaulsBike, numberOfFlatTires, cupsOfCoffeeToday;

```
class FirstProgram {  
    static public void main(String[] args) {  
        System.out.println("Hello world");  
    }  
}  
// Defined a class, but no member variables and only one method
```

Class = Blueprint

One blueprint to create several similar, but different, houses:



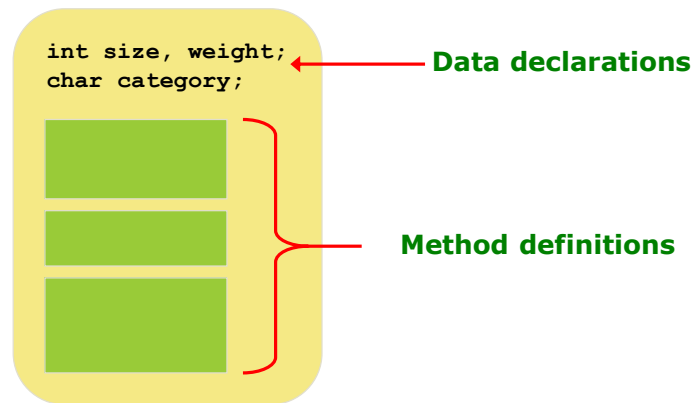
Examples of Classes

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Copyright © 2014 Pearson
Education, Inc.

Classes

A class can contain data declarations and method definitions



Copyright © 2014 Pearson
Education, Inc.

Declare: say something exists, give it a name. Cannot use it until it is defined.

Define: say what something is equal to, give it a value.

Can have data definitions too!

Classes and Objects

An object has *state* and *behavior*

Consider a six-sided die (singular of dice)

- It's state can be defined as which face is showing
- It's primary behavior is that it can be rolled

We represent a die by designing a class called `Die` that models this state and behavior

- The class serves as the blueprint for a die object

We can then instantiate as many die objects as we need for any particular program



Public and Private

Instance variables and class methods can be made:

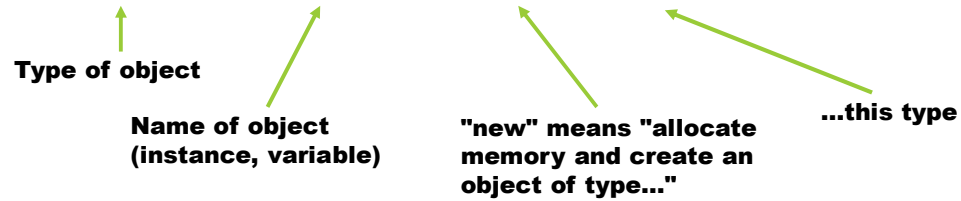
- `public`: code in any class can access them
- `private`: only methods in this same class can access them
- `<default>`: only classes in this same "package" can access them

Do an example with multiple classes!

Syntax: Creating Objects

Particular Java syntax required. Assume class `LibraryItem` is defined:

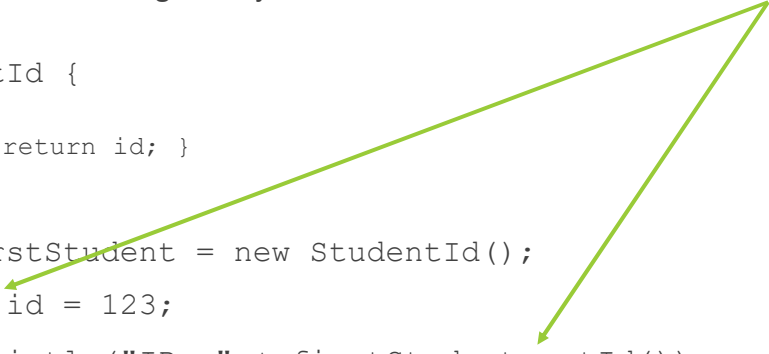
```
LibraryItem javaBasics = new LibraryItem();
```



Syntax: Accessing a Method or Variable

Variables and methods "belong" to objects. How do we call them? Use the "dot" operator

```
class StudentId {  
    int id;  
    int getId() { return id; }  
}  
  
StudentId firstStudent = new StudentId();  
firstStudent.id = 123;  
System.out.println("ID: " + firstStudent.getId());
```



Why have getId() method?

Want to protect underlying variable, probably not make it available to be changed.

Probably have setId() method also: only allow to set 1x, make sure value is legal, etc.

Syntax: Defining Methods

```
Date dueDate(int libraryId) {...} // given libraryId, return due date
```

**Type returned
by method**

**Name of method
(function, procedure)**

**Variables
("parameters",
"arguments"), if any,
passed into method
from calling function**

**Code executed by
the method**

Return type is `void` if no value is returned by the method, e.g.
`public static void main()`

The `return` Statement

The *return type* of a method indicates the type of value that the method sends back to the calling location

A method that does not return a value has a `void` return type

A *return statement* specifies the value that will be returned

```
return expression;
```

expression must conform to the return type

Method Parameters

When a method is called, the *actual parameters* from the caller are copied into the *formal parameters* in the method header

```
ch = obj.calc(2, count, "Hello");
```

```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```


Local Variables

Local variables ("method variables") can be declared inside a method

The formal parameters of a method create *local variables* when the method is invoked

When the method finishes, all local variables are destroyed (including the formal parameters)

Instance variables, declared at the class level, exist as long as the object exists

Instance Variables

A variable declared at the class level (such as `faceValue`) is called an *instance variable* or *class variable*

Class methods can use:

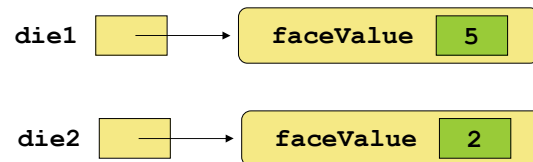
- Their local variables
- Their formal parameters (which become local variables)
- Instance variables

Each instance (object) has its own instance variables

- Each time a `Die` object is created, a new `faceValue` variable is created as well
- This is how two objects with same type can have different states

Instance Variables

We can depict the two `Die` objects from the `RollingDice` program as follows:



Each object maintains its own `faceValue` variable, and thus its own state

```

//*****
// Die.java      Author: Lewis/Loftus
//
// Represents one die (singular of dice) with faces showing values
// between 1 and 6.
//*****

public class Die
{
    private final int MAX = 6; // maximum face value

    private int faceValue; // current value showing on the die

    //-----
    // Constructor: Sets the initial face value.
    //-----
    public Die()
    {
        faceValue = 1;
    }

    continue

```

Copyright © 2014 Pearson
Education, Inc.

First we will look at some code. Then we will explain it.

CONSTRUCTOR: a method called automatically when an object is created.

PUBLIC and PRIVATE: control "access" to methods and variables

continue

```
//-----  
//  Rolls the die and returns the result.  
//-----  
public int roll()  
{  
    faceValue = (int)(Math.random() * MAX) + 1;  
    return faceValue;  
}  
  
//-----  
//  Face value accessor.  
//-----  
public int getFaceValue()  
{  
    return faceValue;  
}
```

continue

```
//-----  
// Returns a string representation of this die.  
//-----  
public String toString()  
{  
    String result = Integer.toString(faceValue);  
  
    return result;  
}
```

```

//*****
// RollingDice.java      Author: Lewis/Loftus
//
// Demonstrates the creation and use of a user-defined class.
//*****

public class RollingDice
{
    //-----
    // Creates two Die objects and rolls them several times.
    //-----
    public static void main(String[] args)
    {
        Die die1, die2;
        int sum;

        die1 = new Die();
        die2 = new Die();

        sum = die1.roll();
        sum += die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
    }
}

```

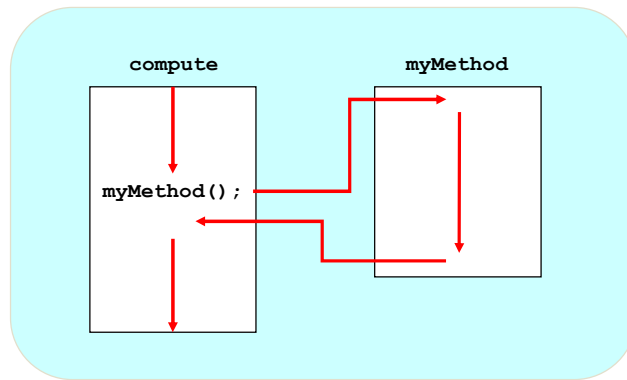
Encapsulation Again

One of the methods would “roll” the die by setting `faceValue` to a random number between one and six

- We **hide** the details of how rolling is implemented
- We **prevent** the user from changing the number of faces, current value, etc

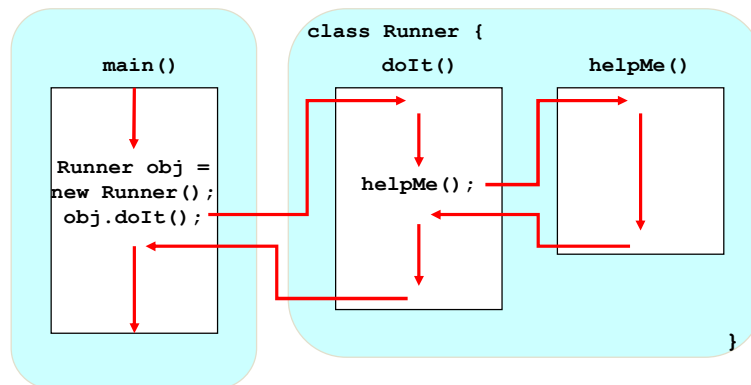
Method Flow of Execution

If the called method is in the same class, only the method name is needed



Method Flow of Execution

Outside of a class's own methods, a class's methods can be called via an object whose type is the class.



The toString() Method

It's good practice to define a `toString()` method for a class

The `toString()` method returns a character string that represents the object in some way

It is called automatically when an object is concatenated to a string or when it is passed to the `println()` method

It's also convenient for debugging problems