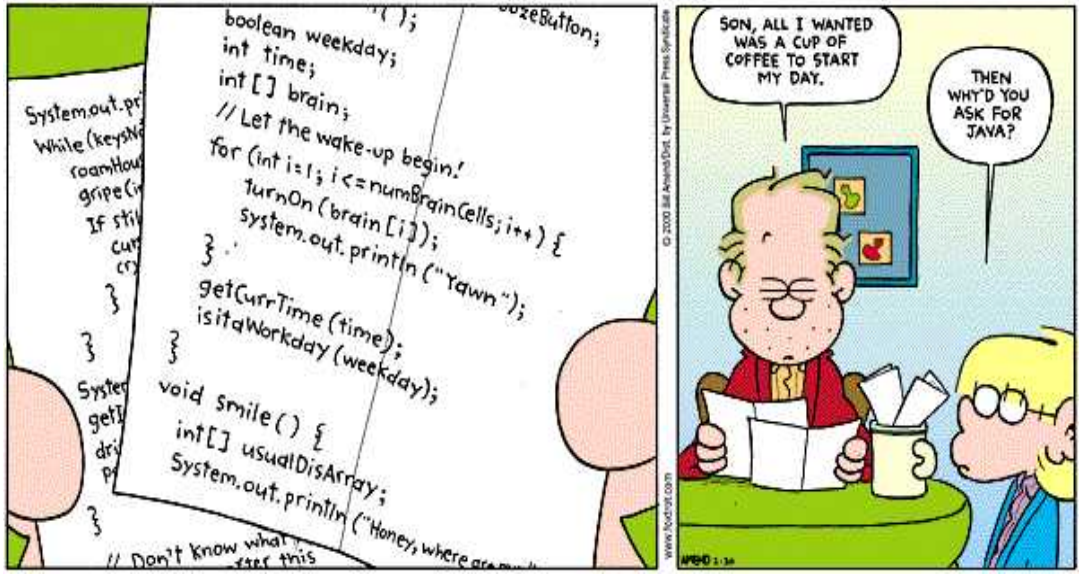
Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Object-Oriented Programming

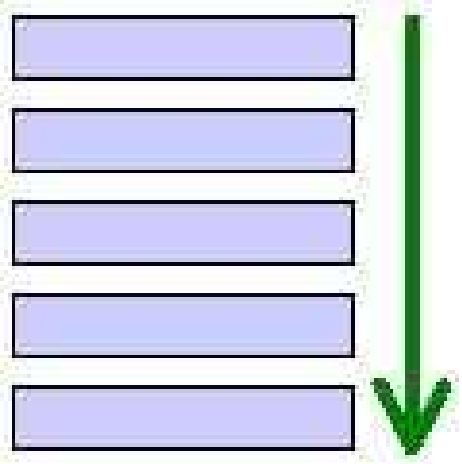


CS2012: Introduction to Programming II

Procedural Programming
vs.
Object-Oriented Programming

Procedural Programming Paradigm

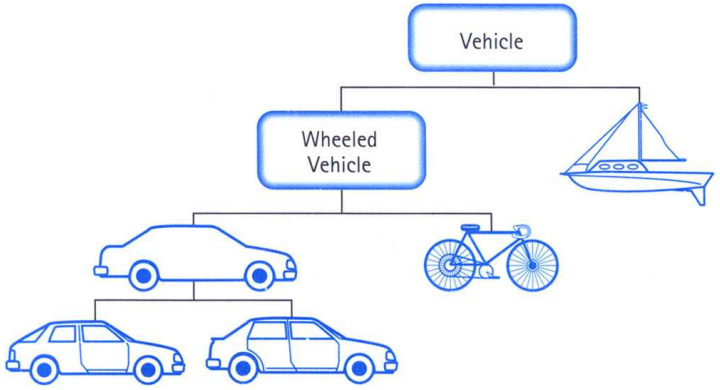
- A series of computational steps to be carried out.
- Any method might be called at any point during a program's execution (even by other methods)
- A list or set of instructions telling a computer what to do step-by-step and how to perform from the first code to the second
- Generally has a top-down, chronological ordering.
- The type of programming you have been doing in CS2011.



Object-Oriented Paradigm

- OOP is VERY different from Procedural
  - requires a shift in thinking and different way to look at programming problems

- Instead of thinking in "steps" to a solution, you need to think about and identify the objects involved in the problem
  - What objects are involved?
  - What information do I need to know about the objects?
  - How do I want the objects to behave?
  - How should the objects interact with one another?



Objects and Classes

Objects

- An object is an instance of a class, and represents a "real world" entity.
  - Tangible Objects:
    - objects you can physically touch
    - Examples: house, car, fan, etc...
  - Non-tangible Objects:
    - objects you can't physically touch
    - can be harder to identify
    - Examples: BankAccount, Insurance, Loan, Vacation, etc...

States

- ***state(s)*** of an object:
  - a.k.a. properties, attributes, data fields, member variables, instance variables
  - all the information you want to store about the object.
  - Example: A `Circle` object with a data field `radius`
  - Example: A `Rectangle` object with data fields `width` and `height`.

Behaviors

- ***behavior(s) of an object***:
  - all actions defined by the methods of the object
  - whenever you invoke (call) an object's methods, you are asking the object to perform some action
  - Example: `Circle` objects could have methods `getArea()`, `getCircumference()`, `setRadius(radius)`

Examples

Object	States	Behaviors
Person	name, age, gender, phone number	get the address, change the phone number, get the name, etc.
Customer	name, address, purchase history	make purchase, list items bought, return item
BankAccount	owner, balance, account number	withdraw, deposit, transfer, get balances

Classes

- Objects of the same type are all defined using a common ***class*** definition.
  - All `Circle` objects would be defined using a `Circle` class.
  - All `CellPhone` objects would be defined using a `CellPhone` class.

- Think of a class like a template, blueprint, or contract.
  - Defines what an object's data fields and methods will be.
  - The data fields (member variables) define the state of the objects.
  - The methods define the behavior of the objects

- Classes also provide special methods called ***constructors*** which are called to construct or create instances of objects from the class template.

Relationship Between Objects and Classes

- An object is an ***instance*** of a class.

- A class is defined **once.**
  - You can create many objects or instances of that class.
  - The terms ***object*** and ***instance*** are interchangeable.
  - Example: I can have five `Circle` objects in my program each created using the `Circle` class.

- ***instantiation:*** creating an instance of a class.

- A good analogy is the relationship between food and its recipe:
  - the recipe to make a cheesecake (yummy!) defines how to create the cheesecake.
  - each cheesecake you make using that recipe is an "instance" or object created from that recipe.

An Object Oriented Example

The `Triangle.java` Class

- The `Triangle` Class defines the logic which is used to construct a `Triangle` object.

- Things to notice:
  - Class header / definition.
  - Access modifiers
  - Data fields
  - Constructors
  - Getters
  - Setters
  - Other Methods
  - toString() Method
- Other Things to Notice:
  - No `main()` method.
    - This class will not execute on its own, even if you try to run it.
  - Absence of `static` keyword in method definitions.
  - use of the `this` keyword

The TriangleMain.java Class

- This is considered the "main" class whose purpose is to test / use Triangle objects created by the Triangle class.

- In object-oriented design, the class in a project which contains the main method can be known as the Tester, Main, or Driver class.
  - Can also be known as the ***client***.
    - Here, client just refers to the part of your project that is using an instance of your Triangle class.

- Recall: A main() method is required to run your program.

Putting Both Together

- Program has two classes now instead of one:
  - `Triangle.java` defines the object we want to use.
  - `TriangleMain.java` contains the main method and is tests our `Triangle` class.

- Object oriented programs are generally made up of multiple classes and one main / driver / tester which starts the program execution.

- NOTE: You can put both classes in the same file, but only one class can be a public class. The public class must also be the same name as the file name. However, it is better to keep each class in a separate file.

Putting Both Together

- Each class is compiled into separate .class files
  - `Triangle.java` compiles to `Triangle.class`
  - `TriangleMain.java` compiles to `TriangleMain.class`.

- All compile .class files are required for your program to execute.

Putting Both Together

- Each instance (object) of the class retains its own data.
  - Each `Triangle` object is independent of the other
  - Changing the data in one object will not affect the data in another.
    - An exception to this rule will be discussed later on.

- You can access the data fields in one of two ways:
  - Directly (only if the data is public) i.e. `triangle1.side1Length;`
  - Through the `getSide1Length()` method i.e. `triangle1.getSide1Length();`
  - The second way is the preferred method since all of your data should generally be **private** (as we will discuss later).

Other Examples

- For other OOP examples, see chapter 09 of your textbook.

Data Fields

Data Fields

- All the data related to an object is stored in its data fields
  - data fields are also known as member data, properties, states, and many others
- Generally listed right after the class header / definition.
- Can be any type:
  - primitive data types
  - array data types
  - even other object data types.
- Have an access modifier:
  - public, private, protected
  - generally all data fields should be private.
- Can be static or non-static
- Have default values
  - default values can be set when the data field is declared.
  - default values can be set in the default constructor.
  - user specified values can be set in other constructors.

The Scope of Data Fields

- Recall: the ***scope*** of a variable is where a variable is visible / can be used in a program.
  - always starts from where a variable is declared and ends at the closing curly brace } of the block which contains the variable
  - a block could be a method, if / else, loop, or even a class block.

- The scope of instance and static data fields is the entire class.
  - They can be declared anywhere inside a class.
  - normally they are declared at the top of the class
  - The exception is when a data field is initialized based on a reference to another data field
    - In this case, the other data field must be declared first

Scope Example

```
public class Circle{
  public double getArea(){
    return radius*radius*Math.PI;
  }

  private double radius=1;
}


 public class Foo {
    private int i;
    private int j=i+1;
 }
```

Object / Reference Type Data Fields

- Data fields can be reference types
- Example: the following `Student` class contains a data field name of the `String` type

```
public class Student {
  String name;                  //default value null
  int age;                      //default value 0
  boolean scienceMajor;         //default value false
  char gender;                  //default value '\u0000'
}
```

- If a reference type data field is not initialized its value is a special literal value, `null`.

Data Field Default Values

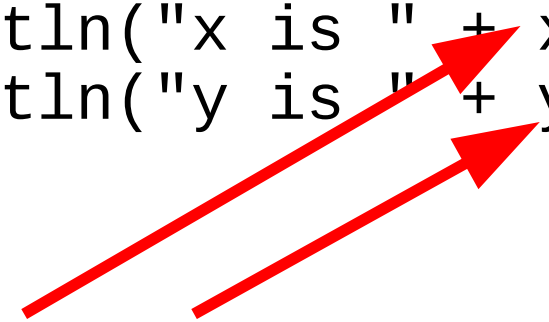
- The default value of a data field is:
  - null for a reference types
  - 0 or 0.0 for numeric types
  - false for a boolean types
  - '\u0000' for a char type

Local Variables and Default Values

- Reminder: ***local variables*** are variables defined inside of a method and are local to that method.
  - NOT NOT NOT the same as data fields.
  - they only exist as long as the method is executing

- Unlike data fields, Java DOES NOT assign default values to local variables.

```
public class Test {
  public static void main(String[] args) {
    int x;     //no default value
    String y;  //no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```



Compilation error: variables not initialized

Constructors

Constructors

- special kind of method with three properties:
  - same name as the class
  - **NO** return type – not even `void`
  - they are invoked using the `new` operator

- Used to create an instance of a class and initialize the data fields of the object.
  - Syntax:
    - `ClassName variableName = new ClassName(arguments);`
  - Example:
    - `Triangle triangle1 = new Triangle(2, 3, 2);`

Constructors

- Constructors can be ***overloaded*** just like regular methods.

- Should generally be public
  - private constructors do exists, but it's rare
  - usually they are just used by another constructor to help initialize the data fields.

- It is very common to do something like the following:

```
public void Circle() { }
```

  - this would not be correct since constructors cannot have a return type.
  - adding a return type turns this into a regular method not a constructor

Private Constructors

- Most of the time constructors should be public.
- In rare cases, you may want to prevent a user from creating an instance of a class.
- This can be accomplished using a private constructor
- Example: there is no reason to create an instance of the Math class since all of its data fields and methods are static. The Math class has the following constructor to prevent instantiating the class:
  - `private Math() { }`

No-Arg Constructor

- Every class should provide a constructor without any arguments.
  - e.g. public Triangle() { }

- Usually used to create instances of the class with default values for each of the data fields.

- Best practice is to always provide one of these unless you have a good reason for not doing so.

The Implicit Default Constructor

- Under certain conditions, the compiler automatically provides a no-arg constructor called the ***default constructor***.

- Rules:
  - if you define a class ***without any constructors***, the compiler will automatically provide a public no-arg constructor with an empty body.
  - if your class defines even one other constructor, the compiler will NOT provide the default constructor even if the no-arg constructor was not defined.

- NOTE: The Default Constructor IS a no-arg constructor, the only difference is if the compiler provides a no-arg constructor it is called the default constructor.

Class Methods

Class Methods

- Classes can have all sorts of other methods which perform operations on the data fields of the class.

- If a method doesn't do ANYTHING to or with the data fields of a class, it more than likely doesn't belong there.

- Methods can return any data type, or be void.

- Methods will be made up of all the techniques you have used so far:
  - variables and calculations
  - if / else statements
  - loops
  - arrays
  - maybe call other methods
  - use other Classes or even instances of the same class.

Packages

Packages

- Packages are used to organize classes

- To include a class in a package, add the following line as the **first non-comment and non-blank statement in the program**
  - `package package_name;`
  - package names should be all lowercase with words separated by underscores.

- If a class is declared without the package statement, the class is placed in the ***default package***.

-

- ADD STUFF ABOUT CROSS PACKAGE INTERACTION

The `this` Keyword

The `this` Reference

- this is a built in keyword which always refers to the (calling object) object itself

- it can only be used inside of a Class definition.
  - see `Triangle.java` for examples.

Using `this` To Access Hidden Data Fields

- Sometimes the parameter of a method is the same name as the data field whose value it is trying to replace.
  - In this case we say the data field is "hidden" within the scope of the method.
  - The only way to refer to this data field is using the `this` keyword

- "hidden" static data fields can be accessed by using Classname.staticVariableName

- Best coding practice dictates that you should ALWAYS use `this` in your class when you are referring to any instance data fields or methods.
  - This helps with better code readibility.

Using `this` to Invoke a Constructor

- this can be used to call another constructor in the same class.
  - see Triangle.java for an example.

- `this(argument-list)` must be the first line in the constructor if you wish to use this technique.

- ProTip: If your class has multiple constructors is is better to implement them using `this(argument-list)` as much as possible. This helps to simplify coding, minimize mistakes, and makes the class easier to read and maintain.

Thoughts on `this`

- Opinions differ on whether or not use of `this` is necessary in all cases.
- When is `this` absolutely required:
  - Accessing “hidden” data fields.
  - Calling a constructor from another constructor.
- Some people believe excessive use of this is too verbose.
- Others feel it can make your code more readable.
  - I mostly fall into this category.
- For your work, I have no preference either way, but most of my examples will use this as much as possible.
  - Whatever you choose, be consistent.
  - Also, some companies will actually have guidelines as to what they feel the proper use of “this” is. Always follow the guidelines of your employer!

Encapsulation

Encapsulation

- Data fields in a class could be declared public.
  - public data fields are directly accessible through a reference variable.

- Example: Assume side1Length is public.

```
Triangle t1 = new Triangle();
t1.side1Length = -10;
```

Encapsulation

- This is very bad in practice:
  - data can be put into an invalid state.
    - sideLength1 should not be a negative number but since it was declared public, anyone using our class could change the value to anything they wanted regardless of how the data field is supposed to function.
  - public data fields make a class more difficult to maintain and makes it vulnerable to all kinds of bugs.

- What if we wanted to go back and modify the Triangle class to ensure that all sides cannot be negative?
  - not only do we have to change Triangle, but we would would also have to change any programs which have used the class because the clients may have directly modified the side length values.

Encapsulation

- To prevent direct modification of data fields, you should declare them to be private.
  - this is called ***data field encapsulation*** or just ***encapsulation***.
  - encapsulation is one of the four pillars of object oriented design.
  - if a data field is private, then it can only be accessed directly within its own class giving you more control over how that data can be altered / accessed.

- How do you modify / retrieve the information stored in a private data field?
  - you provide public getter and setter methods.

Getter / Accessor Methods

- methods created specifically to retrieve the data in private data fields.

- the convention is to have the following method header:
  - `public returnType getDataFieldName()`

- getters with a boolean return type should have the following method header:
  - `public boolean isDataFieldName()`
  - `public boolean hasDataFieldName()`

- generally you will have one getter per data field whose data you want to make accessible (readable).
  - data fields which provide getters are known as ***read*** data fields.

Setter / Mutator Methods

- methods created specifically to allow data fields to be altered in a way that is defined in the method.
- gives you more control over how the data of an object can be altered
- conventionally setters should be defined as:
  - `public void setPropertyName(datatype propertyValue)`
- setters should normally not return any values
- generally have one setter per data field you want to allow to be altered
  - data fields which provide setters are called ***write*** data fields.
- data fields which provide both a getter and setter are known as ***read / write*** data fields.

The toString() Method

toString()

- Suppose to want to display a textual representation of your object.
  - usually contains the values of its member data formatted in a way to look good on the console or GUI output

- Suppose you want to be able to print out the reference variables of your objects much like you would normal variables
  - using `System.out.print()` or `System.out.println()`

- Every Class has a `toString()` method by default.
  - Every class is a sub class of the `Object` class
  - Every class inherits a simple `toString()` method from its parent Object class.
    - NOTE: inheritance will be covered at a later time.
  - However, this version of `toString()` tends to print meaningless information about our object.

`toString()`

- We can implement our own version of `toString()` by replacing the implementation of the original version
  - this is called `overriding` a method
  - this is different from `overloading` a method

- The method header of toString() should ALWAYS be:

```
@Override
public String toString()
```

  - method header must exactly match the above
  - cannot have any parameters
  - must always return a String
  - NOTE: @Override is an annotation. These will be discussed at a later time.

- Once this has been implemented, you can now use normal Java print statements to print out a textual representation of your object.
  - `toString()` is automatically called whenever you print a reference variable of your object.

Objects and Methods

Passing Objects to Methods

- Objects can be passed to methods in the same way that primitive types can be passed to methods.

- Primitive types are passed using **pass-by-value**
  - the value of the argument is copied into the parameter
  - changes to the value inside the method do not change the value outside the method

- Object / Reference types **pass-reference-by-value**
  - a copy of the reference is passed into the parameter.
  - changes to the object inside the method WILL change the object outside the method as well.

Return Objects from Methods

- Methods can also return object types.
- Simply state the reference type as the return type of your method.
- See TriangleMain.java for examples of passing an object to a method and returning an object from a method.