Main OOP Concepts:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

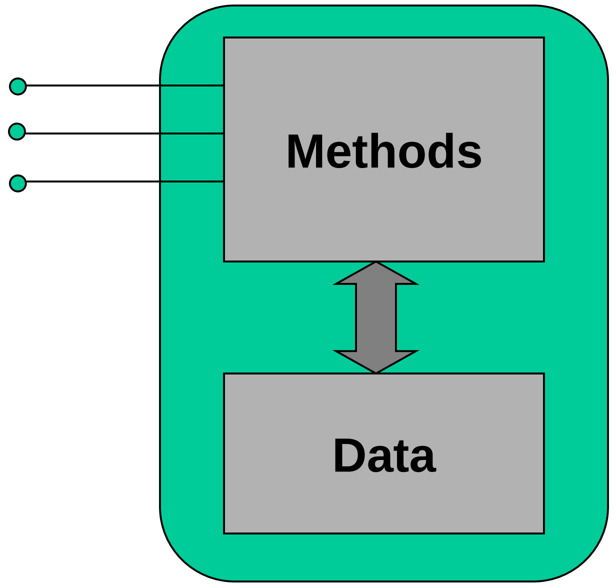
Encapsulation

Encapsulation

- Encapsulation means that all data members (*fields*) of a class are declared <u>private</u>. Some methods may be private, too.
- The class interacts with other classes (called the *clients* of this class) only through the class's constructors and public methods.
- Constructors and public methods of a class serve as the *interface* to class's clients.

What is encapsulation?

- Each object protects and manages its own data. This is called self-governing.



Visibility Modifiers

- We accomplish encapsulation using visibility modifiers.
  - `public`
    - Can be directly referenced from outside of an object.
  - `private`
    - Cannot be referenced externally.
    - Instance data should be defined private.

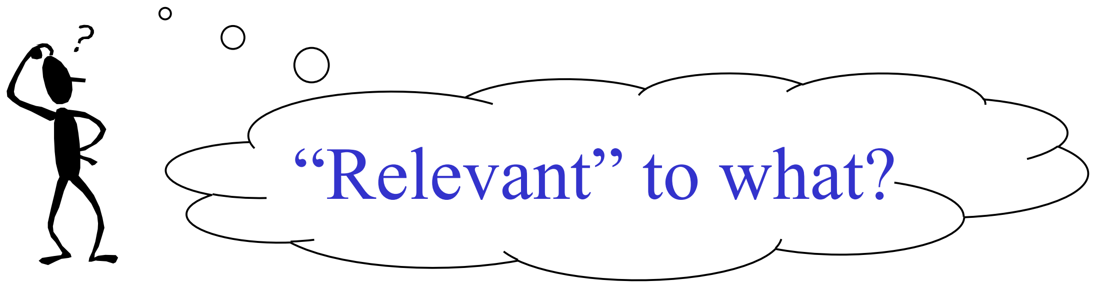
Visibility Effects

	`public`	`private`
`Variables`	Violate encapsulation	Enforce encapsulation
`Methods`	Provide services to outside	Support other methods in class

Abstraction

Abstraction

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones...



- ... relevant to the given project (with an eye to future reuse in similar projects).

Introducing new data types

- To introduce a new data type, we must define its values, data representation, and operations.
- In C#/Java, use a **class declaration**:
  - The class's **instance variables** determine the values and data representation.
  - The class's **constructors** and **methods** are the operations.
- Each object of the class:
  - has those instance variables
  - is created by one of those constructors
  - may be inspected and/or updated by any of those methods.

Public vs. private data representation

- If the data representation is **public**:
  - – Application code might make improper values.
  - – Existing application code might be invalidated by change of representation.
- If the data representation is **private**:
  - + Application code cannot make improper values.
  - + Existing application code cannot be invalidated by change of representation.

Abstract data types

- An **abstract data type** (**ADT**) is characterized by:
  - a set of **values**
  - a set of **operations**.

  It is **not** characterized by its data representation.
- The data representation is **private**, so application code cannot access it. (Only the operations can.)
- The data representation is **changeable**, with no effect on application code. (Only the operations must be recoded.)

ADT specification (1)

- Each ADT should have a **contract** that:
  - specifies the set of values of the ADT
  - specifies each operation of the ADT
    (i.e., the operation’s name, parameter type(s), result type, and observable behavior).
- The contract does **not** specify the data representation, nor the algorithms used to implement the operations.
- The **observable behavior** of an operation is its effect as ‘observed’ by the application code.
  - Example of observable behavior: search an array.
  - Examples of algorithms with that behavior: linear search, binary search.

ADT specification (2)

- The **ADT programmer** undertakes to provide an implementation of the ADT that respects the contract.
- The **application programmer** undertakes to process values of the ADT using only the operations specified in the contract.
- **Separation of concerns**:
  - The ADT programmer is not concerned with what applications the ADT is used for.
  - The application programmer is not concerned with how the ADT is implemented.
- Separation of concerns is essential for designing and implementing large systems.

Example : simplified contract for Space Invaders Defender

- Must be able to die/be hit
- Get the bullet start point
- Move
- Reset

The Defender Class

int kInterval; //distance to travel

bool Died;

int Pointx; //position

int Pointy;

Defender()

MoveLeft()

Moveright()

BeenHit(0

Get bulletStart()

Draw()

Reset()

Implementation By The Abstract Data Type Programmer

```
public class Defender : GameObject
{
            private int kInterval;
            private bool Died;
            private int Pointx;
            private int Pointy;
```

The data type would be represented with data structures

```
            public Defender()
            {
                        Pointy = 200;
                        Pointx = 200;
                        kInterval = 5;
                        Died = false;
                        this.draw;
            }
           public void MoveLeft ()
           {
           Pointx -= kInterval;
           if (Pointx < 0)
            Pointx = 0;
           }
                    ….
```

The methods would be written

Implementation By The Application Programmer

Implementation is by method calls:

```
Defender Ship1 = new defender();

Ship1.MoveLeft();
Ship1.BeenHit();
```

Subsequent Implementation By The Abstract Data Type Programmer

A change to the way a method is implemented.

i.e. the code of the body may change but not the functionality or the way it is called.

A change to the way the data is represented.

e.g. change from using an array to a linked list for storage should not affect the application code.

There will be no need for the applications programmer to subsequently change coding.

The calls and parameters passed remain unchanged as per the contract.

Application code will still run as expected (and perhaps more efficiently thanks to the change).

Inheritance

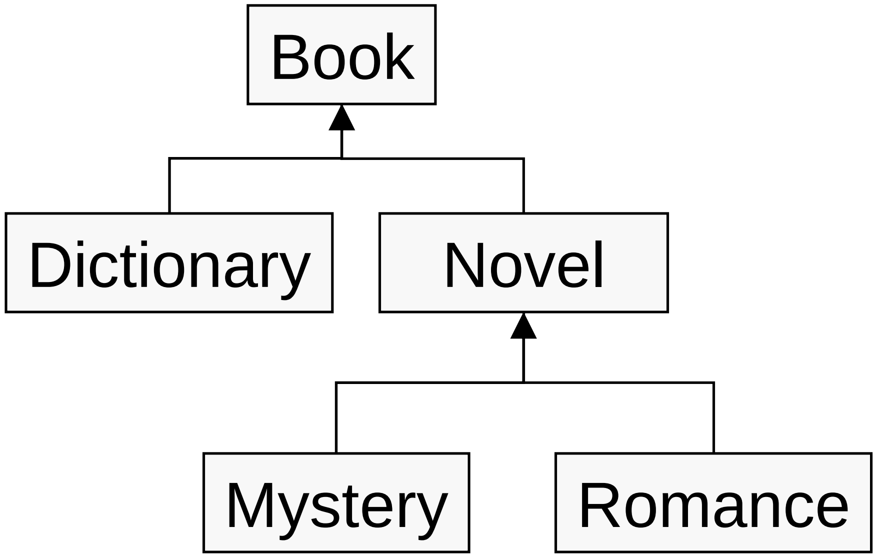
Inheritance

- A class can <u>extend</u> another class, inheriting all its data members and methods while redefining some of them and/or adding its own.
- A class can <u>implement</u> an interface, implementing all the specified methods.
- Inheritance implements the “is a” relationship between objects.

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class* or *superclass*
- The derived class is called the *child class* or *subclass*.
- Creates an is-a relationship

  The subclass is a more specific version of the Original
- (Remember has-a is aggregation.)



Inheritance

- The child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Dictionary extends Book {

        // class contents
}
```

```
Dictionary webster = new Dictionary();

webster.message();

webster.defMessage();
```

```
Number of pages: 1500

Number of definitions: 52500

Definitions per page: 35
```

```
public class Book {

    protected int pages = 1500;

    public String message() {
        System.out.println(“Number of pages: ” + pages);
    }
}

public class Dictionary extends Book {

    private int definitions = 52500;

    public void defMessage() {
        System.out.println(“Number of definitions” +
                           definitions);
        System.out.println(“Definitions per page: ” +
                          (definitions/pages));
    }
}
```

Some Inheritance Details

- An instance of a child class does not rely on an instance of a parent class
  - Hence we could create a Dictionary object without having to create a Book object first

- Inheritance is a one-way street
  - The Book class cannot use variables or methods declared explicitly in the Dictionary class

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child

- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child’s constructor is responsible for calling the parent’s constructor
- The first line of a child’s constructor should use the `super` reference to call the parent’s constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent’s class

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }
}


public class Dictionary Extends Book {

    private int definitions;

    Dictionary(int numPages, int numDefinitions) {

        super(numPages);

        definitions = numDefinitions;
    }
}
```

Overriding Methods

- When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version overrides the parent's version in favor of its own.
  - Signature: method's name along with number, type, and order of its parameters
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

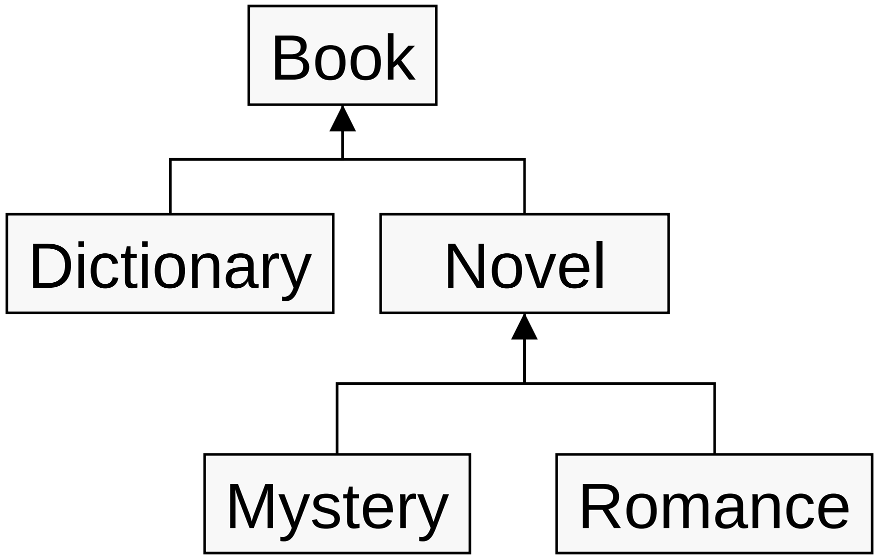
```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }

    public void message() {
        System.out.println("Number of pages: " + pages);
    }
}
public class Dictionary Extends Book {

    protected int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }

    public void message() {
        System.out.println("Number of definitions" +
                              definitions);
        System.out.println("Definitions per page: " +
                              (definitions/pages));
        super.message();
    }
}
```

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*

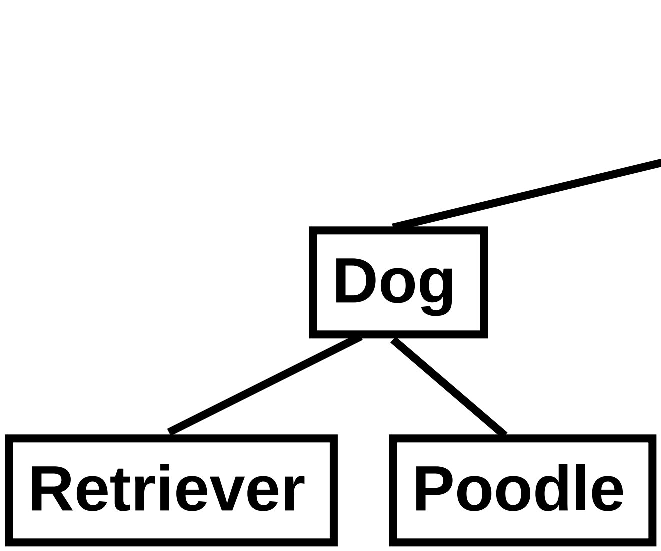


Class Hierarchies

- Two children of the same parent are called *siblings*
  - *However they are not related by inheritance because one is not used to derive another.*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

Polymorphism

A Class Hierarchy



Polymorphism Defined

- The ability to take on different forms.
- Manipulate objects of various classes, and invoke methods on an object without knowing that object’s type.

Polymorphism

- Normally we have this when we create an object:

```
Dog dog = new Dog();
```

- Polymorphism allows us to also do this:

```
Animal pet = new Dog();
```

  - The object reference variable can be a super class of the actual object type! (Does NOT work the other way around: Dog is an Animal but Animal is not necessarily a Dog)

Polymorphic Array Example

```
Animal[] myPets = new Animal[5];
myPets[0] = new Cat();
myPets[1] = new Cat();
myPets[3] = new Dog();

for (int i = 0; i < myPets.length; i++) {
    myPets.makenoise();
}
```

$\}$ You can put any subclass of Animal in the Animal array!

```
Miaow

Miaow

Bark
```

Polymorphic Arguments

```
public class Vet {
    public void giveShot(Animal pet) {
        pet.makeNoise();
    }
}

public class PetOwner {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Cat cat = new Cat();

    vet.giveShot(dog);
    vet.giveShot(cat);
}
```

Polymorphism vs. Inheritance

- Inheritance is required in order to achieve polymorphism (we must have class hierarchies).
  - Re-using class definitions via extension and redefinition

- Polymorphism is not required in order to achieve inheritance.
  - An object of class A acts as an object of class B (an ancestor to A).

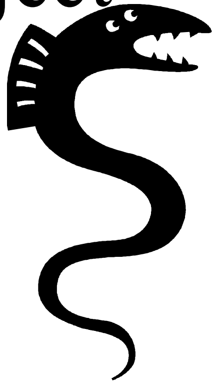
Summary

- **Polymorphism allows objects to represent instances of its own class and any of its sublcasses.**
- **Polymorphic collections are useful for managing objects with common (ancestor) interfaces.**
- **For our purposes, we’ll assume objects “remember” what kind of class they really contain, so method calls are resolved to the original class.**

Abstract Classes

Abstract Classes

- Sometimes we don't want to allow an object to be created of a certain type.
  - What exactly would an Animal object be?
- We use the keyword abstract to prevent a class from ever being instantiated.



```
abstract public class Animal
```

Abstract Classes

- Can still use abstract classes as a reference variable, for the purposes of polymorphism.

- An abstract class has no use until it is extended!

- A class that is not abstract is called concrete.

Abstract Methods

- An abstract method has no body and is marked with the keyword `abstract`.

```
public abstract void eat();
```

- If a method is abstract, the class it is contained in must also be abstract.

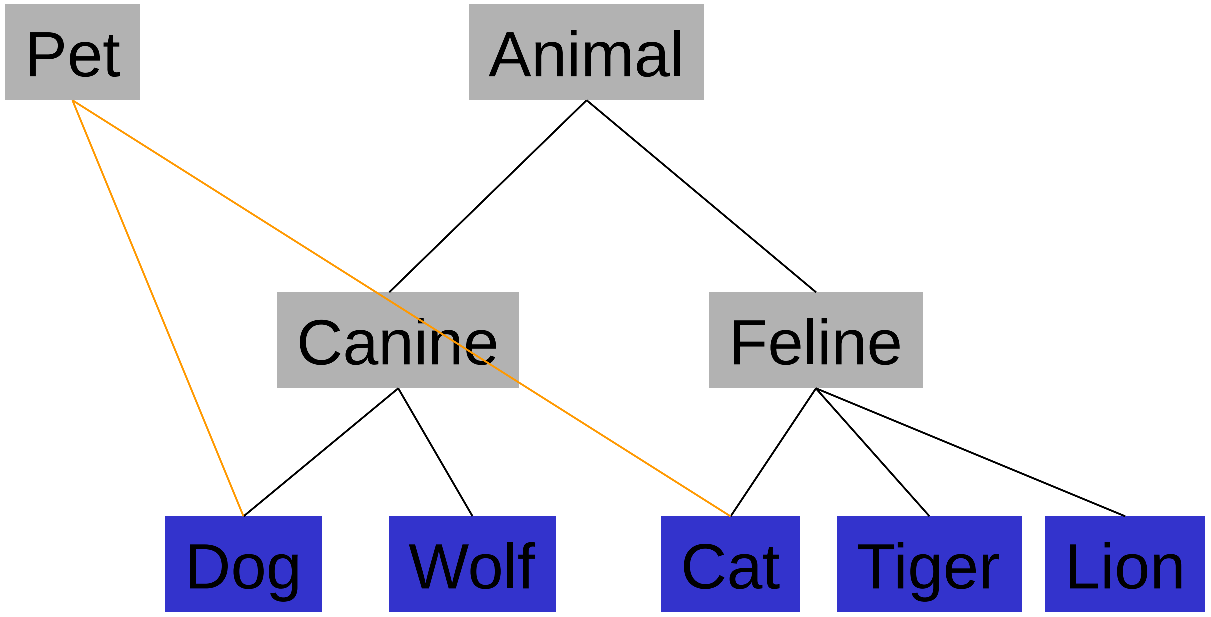
- Abstract methods help the programmer to provide a protocol for a group of subclasses.

- The first concrete class in the inheritance hierarchy must implement the abstract method (i.e. override it and provide it a body)

Interfaces

Multiple Inheritance

- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- Java does not support multiple inheritance



. There is a solution however: **interfaces**.

Interfaces

- Interface: A collection of constants and abstract methods that cannot be instantiated.
- A class implements an interface by providing method implementations for each of the abstract methods defined in the interface.

```
public class Dog extends Canine implements Pet
```

Interfaces

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}

public class Dog extends Canine implements Pet {
    public void beFriendly() {
        wagTail();
    }

    public void play() {
        chaseBall();
    }

    . . .  all the other Dog methods . . .
}
```

Explicitly typing in public and abstract is not necessary since they MUST be public and abstract

Must implement these methods since they are in Pet