Introduction

CSE 2011
Fundamentals of Data Structures

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

- Donald Knuth

Instructor

- Slawomir Kmiec
  - email: skmiec@cse.yorku.ca
  - website: www.cse.yorku.ca/~skmiec
  - Office Hour: LSB103 Wednesdays 6:00 PM - 7:00 PM

Teaching Assistants

- TBD
  - Email: tbd@gmail.com
  - Office Hour: By appointment
- TBD
  - Email: tbd@cse.yorku.ca
  - Office Hour: By appointment

Course Website

- www.cse.yorku.ca/course/2011

Textbook

- Goodrich, M.T. & Tamassia R. (2014). *Data Structures and Algorithms in Java (6th ed.)* John Wiley & Sons.

Summary of Requirements

Component	Weight
4 Assignments	20%
Midterm test (closed book)	30%
Final exam (closed book)	50%

Please see syllabus posted on website for more detailed information.

How to do well in this course

1. Attend all of the lectures!
2. Do the readings **prior** to each lecture.
3. Work hard on each assignment.
   1. Do not leave them to the last moment.
   2. Ask one of the TAs or me if there is something you do not understand.

Course Outline

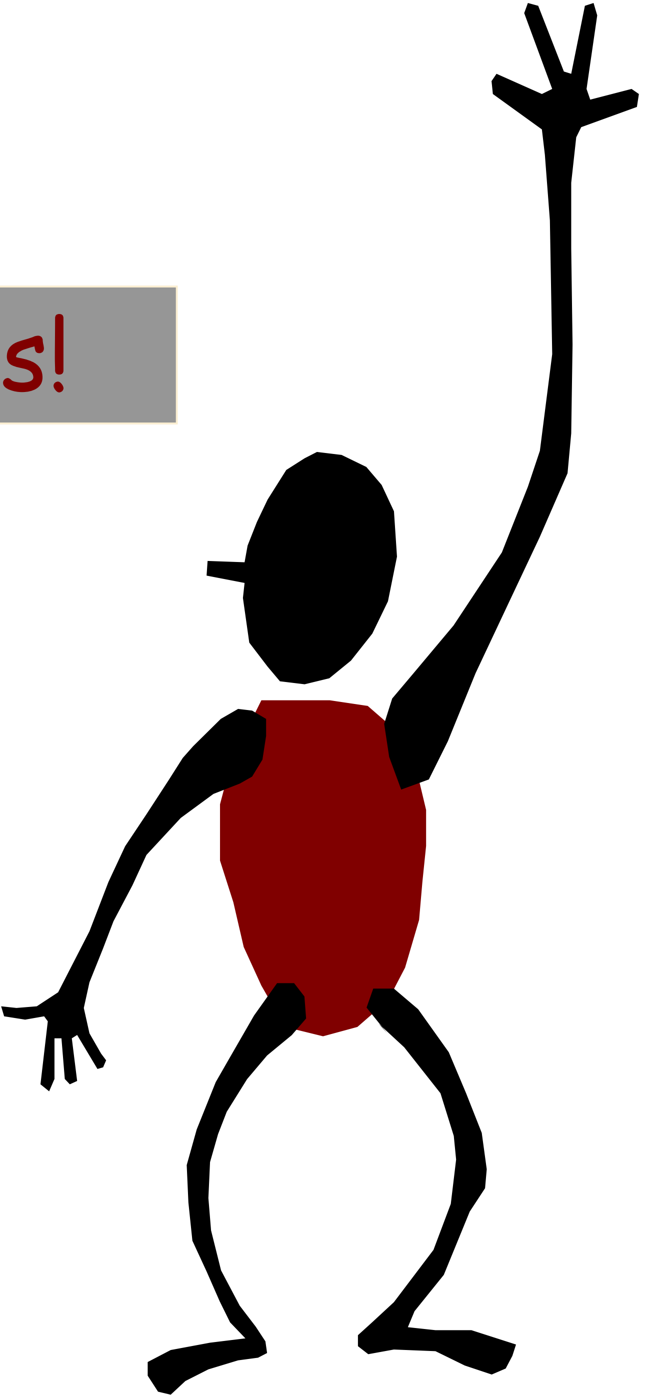
- **Introduction**
- **Analysis Tools**
- **Recursion**
- **Arrays, Array Lists & Stacks**
- **Queues & Linked Lists**
- **Trees**
- **Heaps**
- **Priority Queues**
- **Maps, Hash Tables, Dictionaries**
- **Search Trees**
- **The Java Collections Framework**
- **Graphs**
- **Directed Graphs**
- **Weighted Graphs**
- **Sorting**
- **Graph Search**
- **Strings & Dynamic Programming**

On the slides

- These slides:
  - are posted on the course website.
  - may change up to the last minute as I polish the lecture.
  - Incorporate slides produced by the textbook authors (Goodrich & Tamassia).

Please ask questions!

Help me know what people are not understanding!



Lecture 1

Data Structures and Object-Oriented Design

Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code

Data Structures & Object-Oriented Design

- **Definitions**
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Interfaces
- Casting
- Generics
- Pseudo-Code

Programs = Data Structures + Algorithms

Principles of Object Oriented Design

- Programs consist of objects.
- Objects consist of
  - Data structures
  - Algorithms to construct, access and modify these structures.

Data Structure

- **Definition:** An organization of information, usually in memory, such as a *queue*, *stack*, *linked list*, *heap*, *dictionary*, or *tree*.

Algorithm

- **Definition:** A finite set of unambiguous instructions performed in a prescribed sequence to achieve a goal, especially a mathematical rule or procedure used to compute a desired result.
  - The word *algorithm* comes from the name of the 9th century Persian mathematician Muhammad ibn Mūsā al-Khwārizmī.
  - He worked in Baghdad at the time when it was the centre of scientific studies and trade.
  - The word *algorism* originally referred only to the rules of performing arithmetic using Arabic numerals but evolved via European Latin translation of al-Khwarizmi's name into *algorithm* by the 18th century.
  - The word evolved to include all definite procedures for solving problems or performing tasks.

Data Structures We Will Study

- Linear Data Structures
  - Arrays
  - Linked Lists
  - Stacks
  - Queues
  - Priority Queues
- Non-Linear Data Structures
  - Trees
  - Heaps
  - Hash Tables
  - Search Trees
- Graphs
  - Directed Graphs
  - Weighted Graphs

Some Algorithms We Will Study

- Searching
- Sorting
- Graph Search
- Dynamic Programming

Please see syllabus posted on website for detailed schedule (tentative).

Design Patterns

- A template for a software solution that can be applied to a variety of situations.
- Main elements of solution are described in the abstract.
- Can be specialized to meet specific circumstances.
- Example algorithm design patterns:
  - Recursion
  - Divide and Conquer

Data Structures & Object-Oriented Design

- Definitions
- **Principles of Object-Oriented Design**
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code

Software Engineering

- Software must be:
  - Readable and understandable
    - Allows correctness to be verified, and software to be easily updated.
  - Correct and complete
    - Works correctly for all expected inputs
  - Robust
    - Capable of handling unexpected inputs.
  - Adaptible
    - All programs evolve over time. Programs should be designed so that re-use, generalization and modification is easy.
  - Portable
    - Easily ported to new hardware or operating system platforms.
  - Efficient
    - Makes reasonable use of time and memory resources.

Premature Optimization

- *Premature optimization is the root of all evil.*
  - Donald Knuth

Premature Optimization

- In general we want programs to be efficient. But:
  - Obviously it is more important that they be correct.
  - It is often more important that they be
    - Understandable
    - Easily adapted
  - In striving for efficiency, it is easy to:
    - Introduce bugs
    - Make the program incomprehensible
    - Make the program very specific to a particular application, and thus hard to adapt or generalize.

Asymptotic Analysis

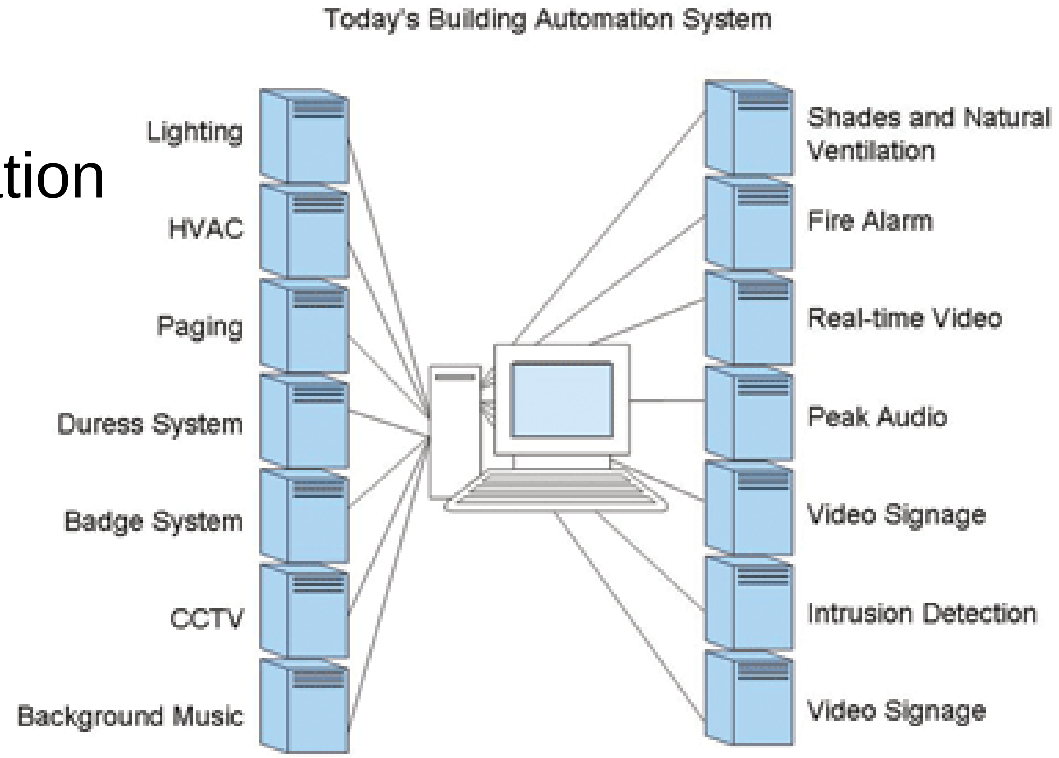
- Asymptotic analysis is a general method for categorizing the efficiency of algorithms.
- Asymptotic analysis helps to distinguish efficiencies that are important from those that may be negligable.
- This will help us to balance the goal of efficiency with other goals of good design.
- This will be the topic of **Lecture 2**.

Principles of Object Oriented Design

- Object oriented design facilitates:
  - Debugging
  - Comprehensibility
  - Software re-use
  - Adaptation (to new scenarios)
  - Generalization (to handle many scenarios simultaneously)
  - Portability (to new operating systems or hardware)

Principles of Object-Oriented Design

- Abstraction
- Encapsulation
- Modularity
- Hierarchical Organization



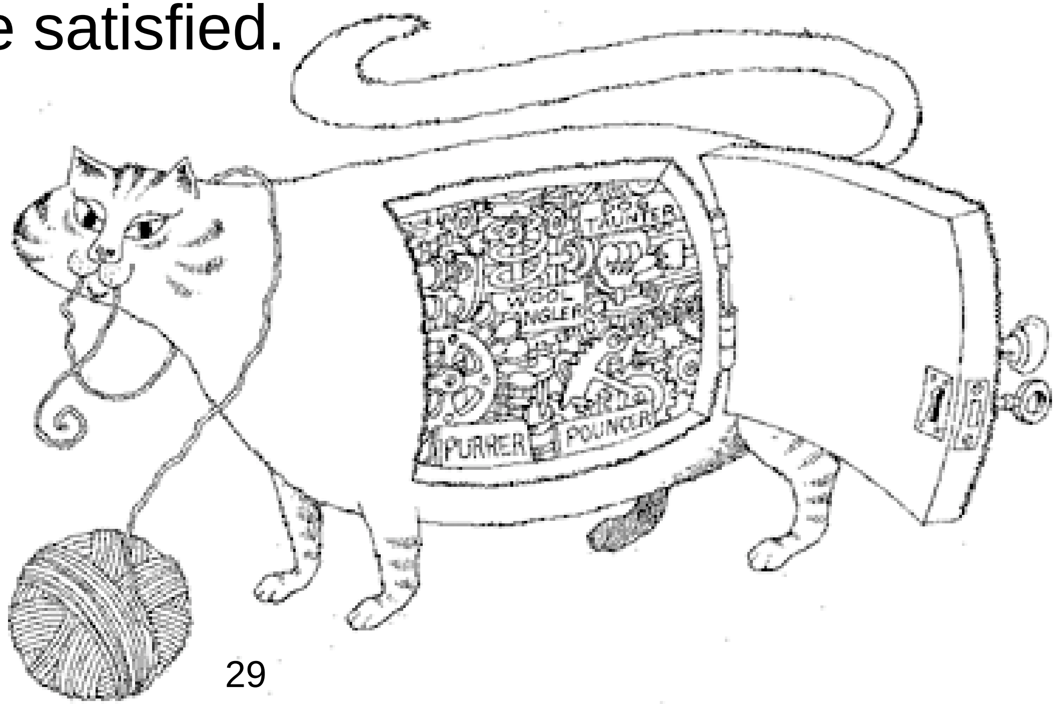
Abstraction

- *The psychological profiling of a programmer is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.*
  - Donald Knuth

Wassily Kandinsky (Russian, 1866-1944)
Abstraction, 1922, Lithograph from the fourth Bauhaus portfolio

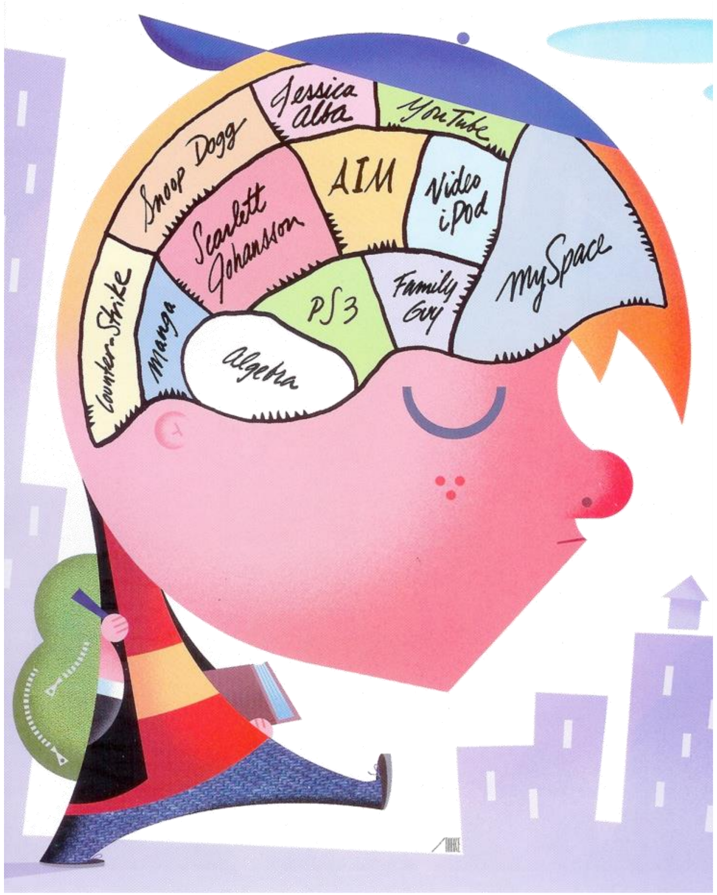
Encapsulation

- Each object reveals only what other objects need to see.
- Internal details are kept private.
- This allows the programmer to implement the object as she or he wishes, as long as the requirements of the abstract interface are satisfied.



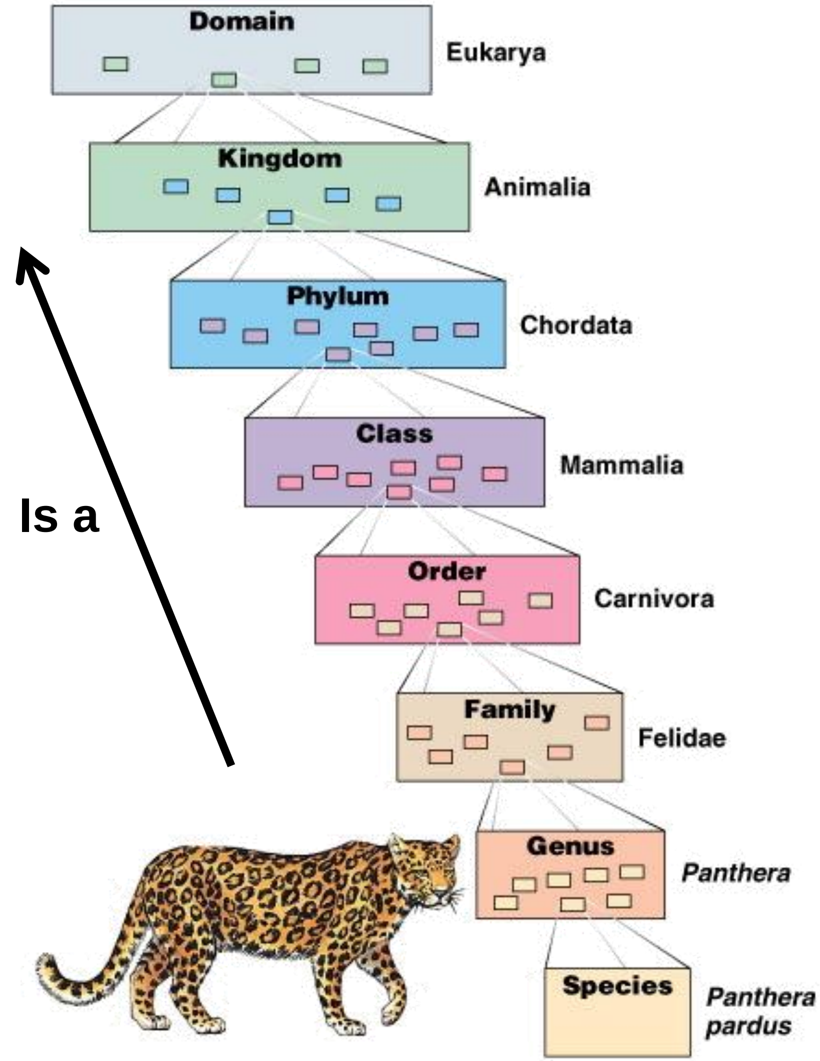
Modularity

- Complex software systems are hard to conceptualize and maintain.
- This is greatly facilitated by breaking the system up into distinct modules.
- Each module has a well-specified job.
- Modules communicate through well-specified interfaces.



Hierarchical Design

- Hierarchical class definitions allows efficient re-use of software over different contexts.



Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- **Hierarchical Design in Java**
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code

Inheritance

- Object-oriented design provides for hierarchical classes through the concept of **inheritance**.
- A **subclass specializes** or **extends** a **superclass**.
- In so doing, the subclass **inherits** the variables and methods of the superclass.
- The subclass may **override** certain superclass methods, **specializing** them for its particular purpose.
- The subclass may also define additional variables and methods that **extend** the definition of the superclass.

Types of Method Overriding

- Generally methods of a subclass **replace** superclass methods.
- An exception is **constructor** methods, which do not replace, but **refine** superclass constructor methods.
- Thus invocation of a constructor method starts with the highest-level class, and proceeds down the hierarchy to the subclass of the object being instantiated.
- This is either accomplished implicitly, or explicitly with the **super** keyword.

Refinement Overriding

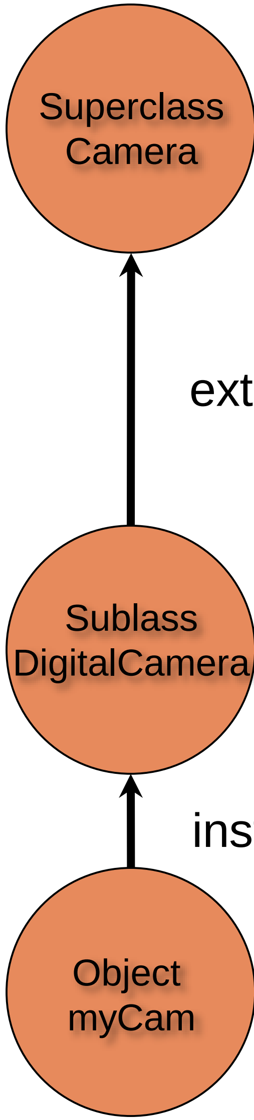
```java
public class Camera {
    private String cameraMake;
    private String cameraModel;

    Camera(String mk, String mdl) { //constructor
        cameraMake = mk;
        cameraModel = mdl;
    }

    public String make() { return cameraMake; }
    public String model() { return cameraModel; }
}
```

```java
public class DigitalCamera extends Camera{
    private int numPix;

    DigitalCamera(String mk, String mdl, int n) { //constructor
        super(mk, mdl);
        numPix = n;
    }

    public int numberOfPixels() { return numPix; }
}
```

refines

extends

```java
DigitalCamera myCam = new DigitalCamera("Nikon", "D90", 12000000);
```



Refinement Overriding

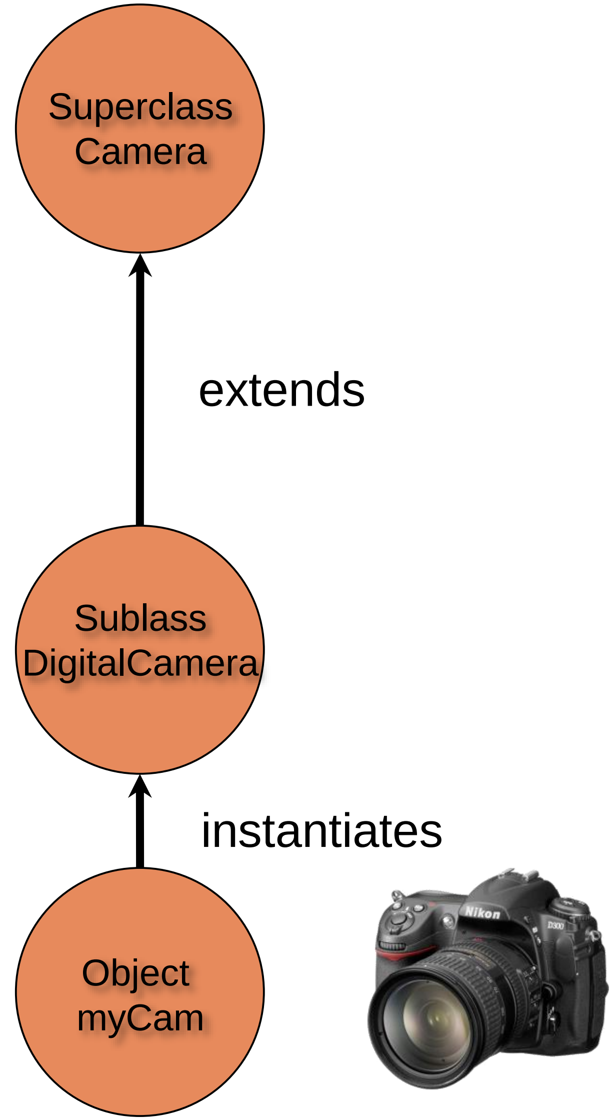
```
public class Camera {
    private String cameraMake;
    private String cameraModel;

    Camera(String mk, String mdl) { //constructor
        cameraMake = mk;
        cameraModel = mdl;
    }

    public String make() { return cameraMake; }
    public String model() { return cameraModel; }
}
```

```
public class DigitalCamera extends Camera{
    private int numPix;

    DigitalCamera(String mk, String mdl) { //constructor
        super(mk, mdl);
        numPix = 0;
    }

    public int numberOfPixels() { return numPix; }
}
```

refines

extends

```
DigitalCamera myCam = new DigitalCamera("Nikon", "D90”);
```



Refinement Overriding

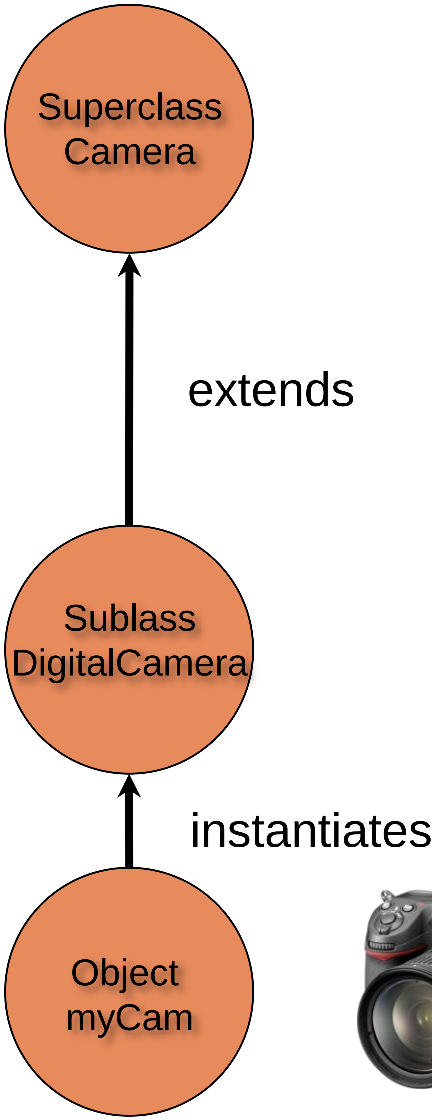
```
public class Camera {
    private String cameraMake;
    private String cameraModel;

    Camera() { //constructor
        cameraMake = "Unknown make";
        cameraModel = "Unknown model";
    }

    public String make() { return cameraMake; }
    public String model() { return cameraModel; }
}
```

```
public class DigitalCamera extends Camera{
    private int numPix;

    DigitalCamera() { //constructor
        numPix = 0;
    }

    public int numberOfPixels() { return numPix; }
}
```

refines (implicit super() call)

extends

```
DigitalCamera myCam = new DigitalCamera();
```



Replacement Overriding

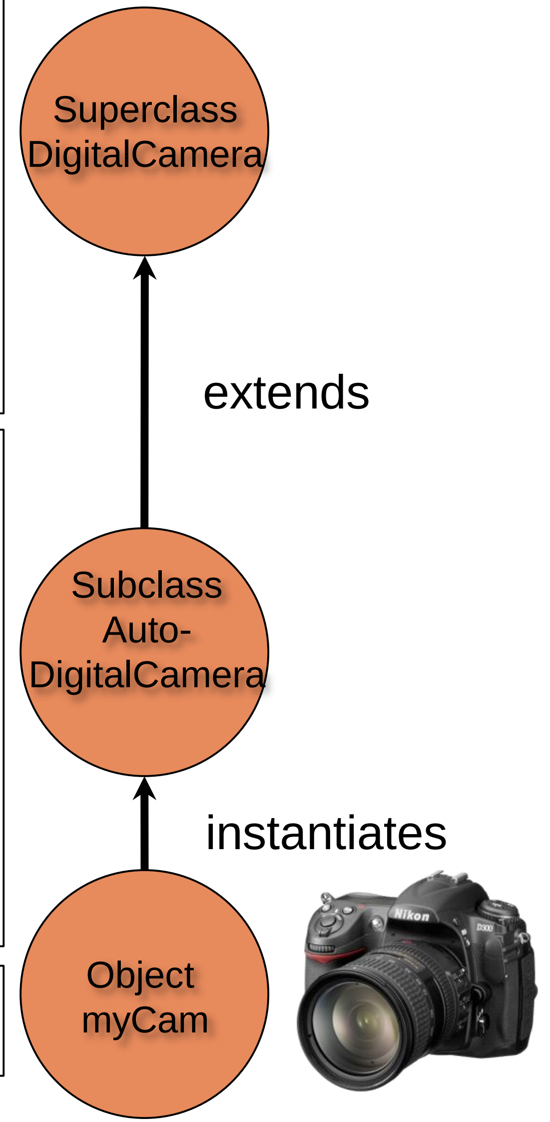
```
public class DigitalCamera extends Camera{
    private int numPix;

    DigitalCamera(String mk, String mdl, int n) { //constructor
        super(mk, mdl);
        numPix = n;
    }

    public int numberOfPixels() { return numPix; }
    public byte[][][] getDigitalImage() { return takeDigitalPhoto(); }
}
```

```
public class AutoDigitalCamera extends DigitalCamera{

   AutoDigitalCamera(String mk, String mdl, int n) { //constructor
        super(mk, mdl, n);
   }

   public byte[][][] getDigitalImage() {
      autoFocus();
      return takeDigitalPhoto();
   }
}
```

replaces

```
DigitalCamera myCam = new AutoDigitalCamera("Nikon", "D90", 12000000);
byte[][][] myImage = myCam.getDigitalImage();
```

polymorphism



Polymorphism

- Polymorphism = “many forms”
- Polymorphism allows an object variable to take different forms, depending upon the specific class of the object it refers to.
- Suppose an object **o** is defined to be of class **S**.
- It is now valid to instantiate **o** as an object of any type **T** that extends **S**.
- Thus **o** can potentially refer to a broad variety of objects.

Replacement Overriding

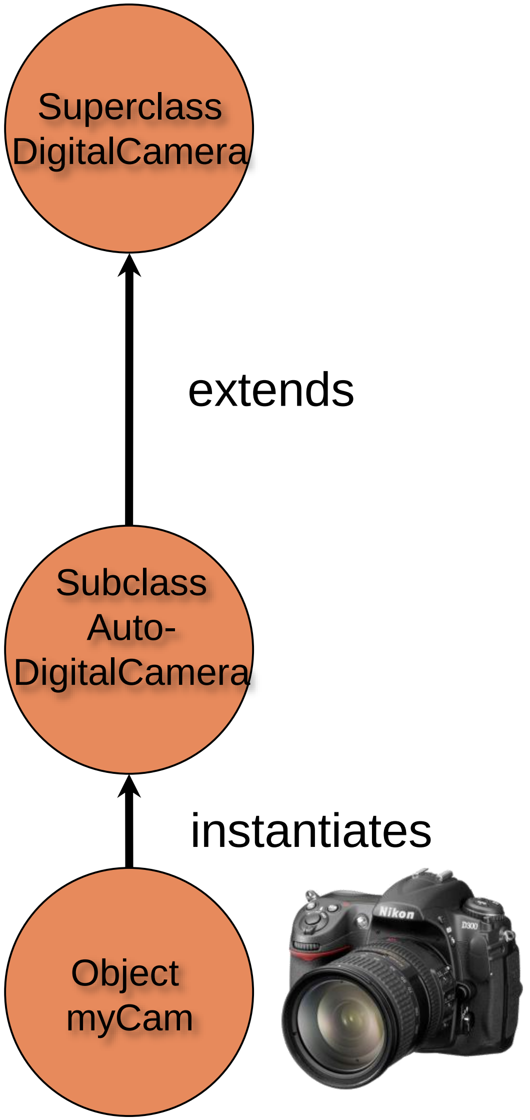
```
public class DigitalCamera extends Camera{
    private int numPix;

    DigitalCamera(String mk, String mdl, int n) { //constructor
        super(mk, mdl);
        numPix = n;
    }

    public int numberOfPixels() { return numPix; }
    public byte[][][] getDigitalImage() { return takeDigitalPhoto(); }
}
```

```
public class AutoDigitalCamera extends DigitalCamera{

   AutoDigitalCamera(String mk, String mdl, int n) { //constructor
        super(mk, mdl, n);
   }

   public byte[][][] getDigitalImage() {
      autoFocus();
      return takeDigitalPhoto();
   }
}
```

replaces

```
DigitalCamera myCam = new AutoDigitalCamera("Nikon", "D90", 12000000);
byte[][][] myImage = myCam.getDigitalImage();
```

polymorphism



Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- **Abstract Data Types & Interfaces**
- Casting
- Generics
- Pseudo-Code

Abstract Data Type (ADT)

- A set of data values and associated operations that are precisely specified independent of any particular implementation.
- ADTs specify what each operation does, but not how it does it.
- ADTs simplify the design of algorithms.

Abstract Data Type (ADT)

- In Java, an ADT
  - can be expressed by an **interface**.
  - is realized as a complete data structure by a **class**.
  - is instantiated as an **object**.

Application Programming Interfaces (APIs)

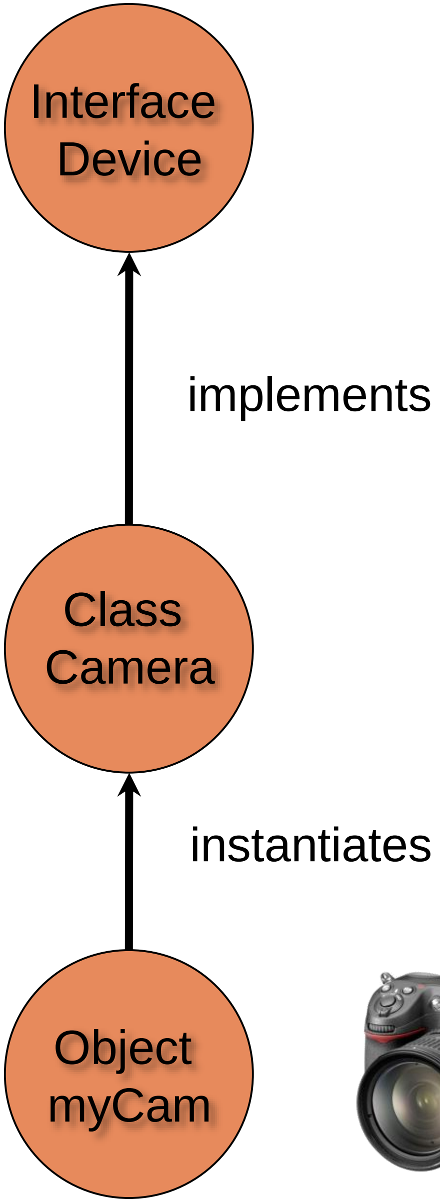
- The **interface** for an ADT specifies:
  - A type definition
  - A collection of methods for this type
  - Each method is specified by its **signature**, comprising
    - The name of the method
    - The number and type of the arguments for each method.

ADT Example

```
public interface Device {
   public String make();
   public String model();
}
```

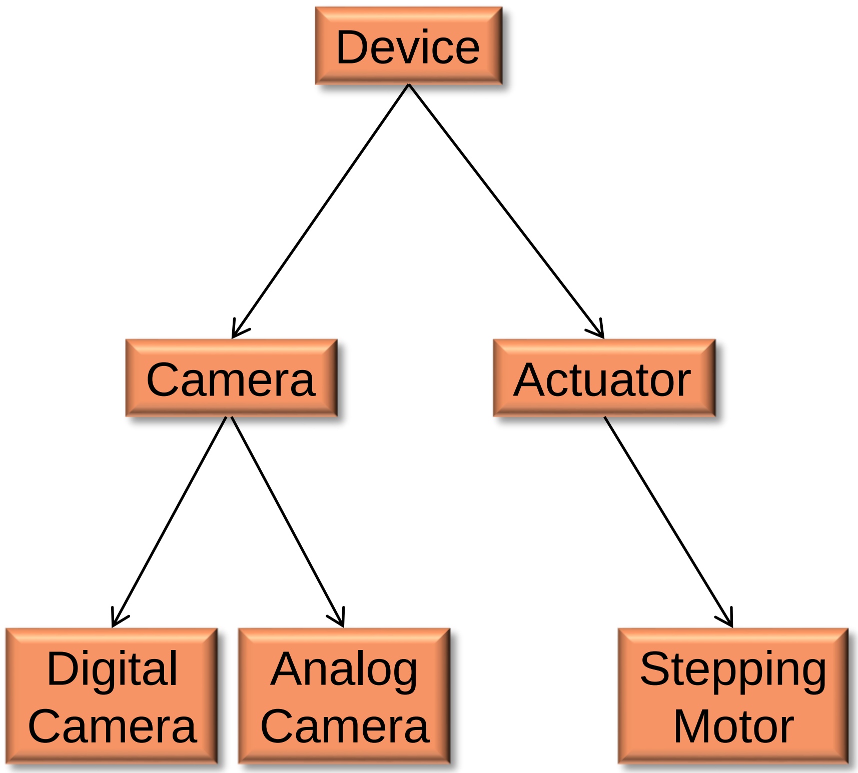
```
public class Camera implements Device {

    private String cameraMake;
    private String cameraModel;
    private int numPix;

    Camera(String mk, String mdl, int n) { //constructor
        cameraMake = mk;
        cameraModel = mdl;
        numPix = n;
    }

    public String make() { return cameraMake; }
    public String model() { return cameraModel; }
    public int numberOfPixels() { return numPix; }
}
```

```
    Camera myCam = new Camera("Nikon", "D90", 12000000);
```



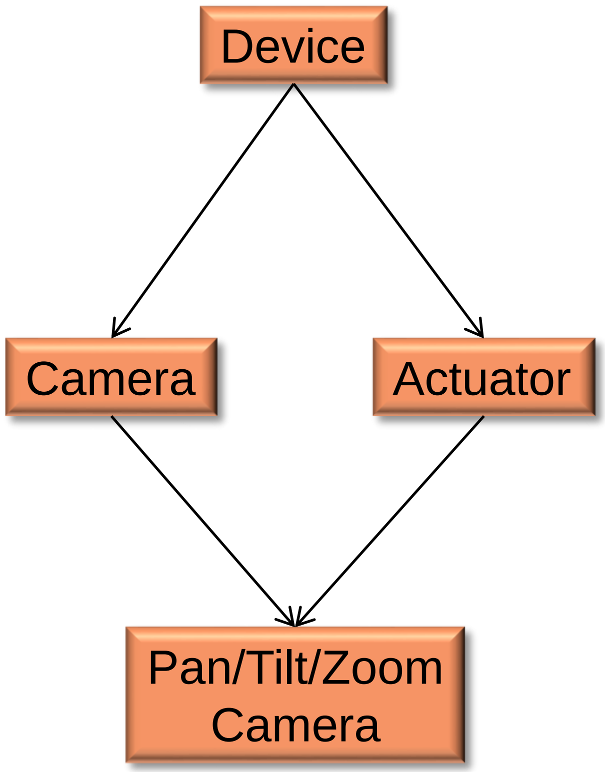
Multiple Inheritance

- In Java, a class can have at most one direct parent class.
- Thus classes must form trees.
- This avoids the ambiguity that would arise if two parents defined methods with the same signature.



Multiple Inheritance

- However, **interfaces** can have more than one direct parent.
- Thus interfaces do not necessarily form trees, but **directed acylic graphs (DAGs)**.
- No ambiguity can arise, since methods with the same signature can be considered as one.
- This allows **mixin** of unrelated interfaces to form more complex ADTs.




```
public interface PTZCamera extends Camera, Actuator {
        …
}
```

Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- **Casting**
- Generics
- Pseudo-Code

Casting

- Casting may involve either a **widening** or a **narrowing** type conversion.
- A **widening conversion** occurs when a type T is converted into a ‘wider’ type U.
  – Widening conversions are performed automatically.
- A **narrowing conversion** occurs when a type T is converted into a ‘narrower’ type U.
  – Narrowing conversions require an explicit cast.

Casting Examples

```
DigitalCamera myCam1 = new DigitalCamera("Nikon","D90");
DigitalCamera myCam2 = new AutoDigitalCamera("Olympus","E30",12000000);
AutoDigitalCamera myCam3 = new AutoDigitalCamera("Sony","A550",14000000);

myCam1 = myCam3; ?
myCam3 = myCam1; ?
myCam3 = myCam2; ?
myCam3 = (AutoDigitalCamera) myCam1; ?
myCam3 = (AutoDigitalCamera) myCam2; ?
```

Casting Examples

```
DigitalCamera myCam1 = new DigitalCamera("Nikon","D90");
DigitalCamera myCam2 = new AutoDigitalCamera("Olympus","E30",12000000);
AutoDigitalCamera myCam3 = new AutoDigitalCamera("Sony","A550",14000000);

myCam1 = myCam3; //widening conversion
myCam3 = myCam1; //compiler error
myCam3 = myCam2; //compiler error
myCam3 = (AutoDigitalCamera) myCam1; //run-time exception
myCam3 = (AutoDigitalCamera) myCam2; // valid narrowing conversion
```

Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- **Generics**
- Pseudo-Code

Generics

- A **generic type** is a type that is not defined at compilation time.
- A generic type becomes fully defined only **when instantiated as a variable**.
- This allows us to define a class in terms of a set of **formal type parameters**, that can be used to abstract certain variables.
- Only when instantiating the object, do we specify the **actual type parameters** to be used.

Generics Example

```
/** Creates a coupling between two objects */
public class Couple<A, B> {
   A obj1;
   B obj2;

   public void set(A o1, B o2) {
       obj1 = o1;
       obj2 = o2;
   }
}
```

```
Camera myCam1 = new DigitalCamera("Nikon","D90”,12000000);
Camera myCam2 = new AutoDigitalCamera("Olympus","E30",12000000);

Couple<Camera,Camera> stereoCamera = new Couple<Camera,Camera>();

stereoCamera.set(myCam1, myCam2);
```

Generics Example

```
Couple<Camera,Camera> stereoCamera = new Couple<Camera,Camera>();
```

- Note that two things are happening here:
  1. The variable stereoCamera is being defined of type Couple<Camera, Camera>
  2. An object of type Couple<Camera, Camera> is created and stereoCamera is updated to refer to that object.

Inheritance with Generics

```
Couple<Camera,Camera> stereoCamera = new Couple<Camera,Camera>();
```

- Generic classes can serve as superclasses or subclasses of other generic and non-generic classes.
- Thus, for example, if a class **CloselyCouple<T, T>** is defined to extend Couple<T, T>, then it would be valid to instantiate stereoCamera as:

```
Couple<Camera,Camera> stereoCamera = new CloselyCouple<Camera,Camera>();
```

But be careful…

- DigitalCamera is a subclass of Camera.
- This does **NOT** mean that Couple<DigitalCamera, DigitalCamera> is a subclass of Couple<Camera, Camera>.
- Thus

```
Couple<Camera,Camera> stereoCamera = new Couple<DigitalCamera,DigitalCamera>();
```

or

```
Couple<Camera,Camera> stereoCamera = new CloselyCouple<DigitalCamera,DigitalCamera>();
```

generate compile errors.

Subtyping with Wildcards

- In order to obtain this kind of subtyping with generics, you can use **wildcards.**
- For example:

```
Couple<? extends Camera, ? extends Camera> stereoCamera
= new Couple<DigitalCamera,DigitalCamera>();
```

or

```
Couple<? extends Camera, ? extends Camera> stereoCamera
= new CloselyCouple<DigitalCamera,DigitalCamera>();
```

Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- **Pseudo-Code**

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

  currentMax | ***A***[0]
  for ***i*** | 1 **to** ***n*** - 1 **do**
    if ***A***[*i*] > ***currentMax*** **then**
      currentMax | ***A***[***i***]
  return ***currentMax***

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  var.method (***arg*** [, ***arg***…])

- Return value

  return ***expression***

- Expressions

  | Assignment (like = in Java)

  $=$ Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code