CMSC 132: Object-Oriented Programming II

Java Support for OOP

Department of Computer Science
University of Maryland, College Park

Object Oriented Programming (OOP)

- OO Principles
  - Abstraction
  - Encapsulation
- Abstract Data Type (ADT)
  - Implementation independent interfaces
  - Data and operations on data
- Java
  - Many language features supporting OOP

Overview

- Objects & class, this, super
- References, alias, levels of copying
- Constructor, initialization block
- Garbage collection, destructor
- Package, scope, inner classes
- Modifiers
  - Public, Private, Protected
  - Static, Final, Abstract
- Generic programming

Object & Class

- Object
  - Abstracts away (data, algorithm) details
  - Encapsulates data
  - Instances exist at run time

- Class
  - Blueprint for objects (of same type)
  - Exists at compile time

“this” Reference

- **Description**
  - **Reserved keyword**
  - **Refers to object through which method was invoked**
  - **Allows object to refer to itself**
  - **Use to refer to instance variables of object**

“this” Reference – Example

```
class Node {
    value val1;
    value val2;
    void foo(value val2) {
        … = val1;            // same as this.val1 (implicit this)
        … = val2;            // parameter to method
        … = this.val2;       // instance variable for object
        bar( this );         // passes reference to object
    }
}
```

Inheritance

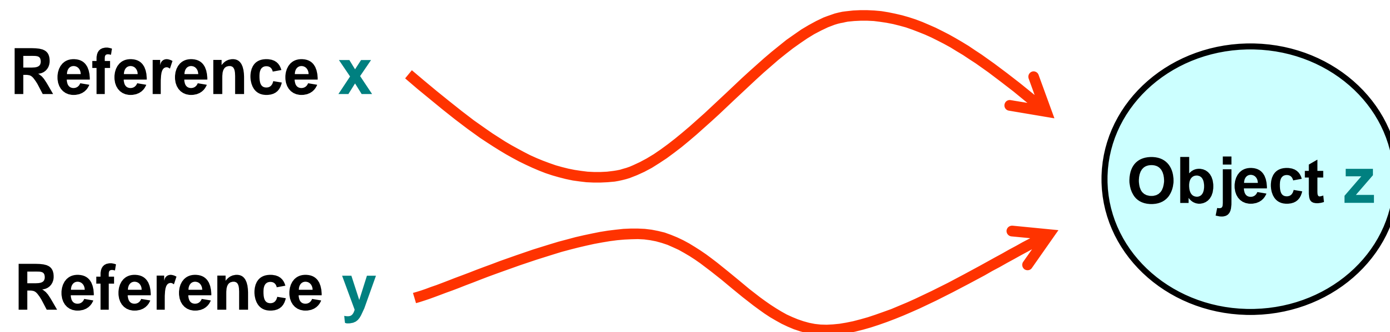
- Definition
  - Relationship between classes when state and behavior of one class is a subset of another class
- Terminology
  - Superclass / parent ⇒ More general class
  - Subclass ⇒ More specialized class
- Forms a class hierarchy
- Helps promote code reuse

“super” Reference

- Description
  - Reserved keyword
  - Refers to superclass
  - Allows object to refer to methods / variables in superclass
- Examples
  - super.x // accesses variable x in superclass
  - super() // invokes constructor in superclass
  - super.foo() // invokes method foo() in superclass

References & Aliases

- Reference
  - A way to get to an object, not the object itself
  - All variables in Java are references to objects
- Alias
  - Multiple references to same object
  - “x == y“ operator tests for alias
  - x.equals(y) tests contents of object (potentially)



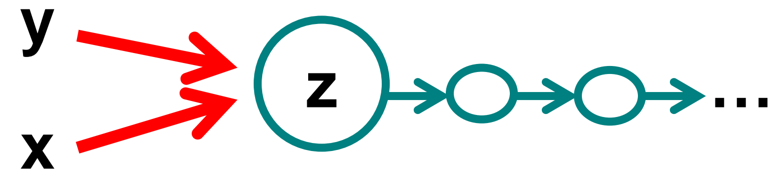
Cloning

- Cloning
  - Creates identical copy of object using clone( )
- Cloneable interface
  - Supports clone( ) method
  - Returns copy of object
    - Copies all of its fields
    - Does not clone its fields
    - Makes a shallow copy

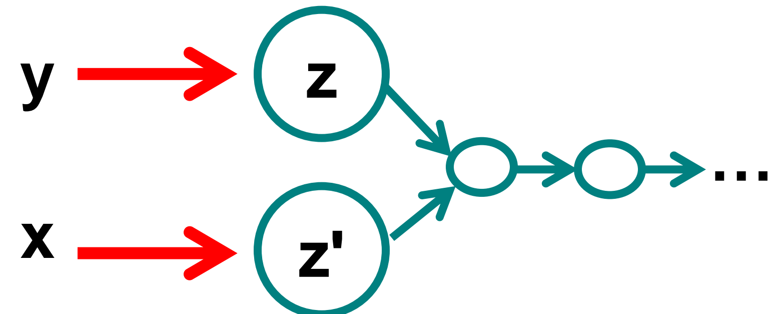
Three Levels of Copying Objects

- Assume y refers to object z

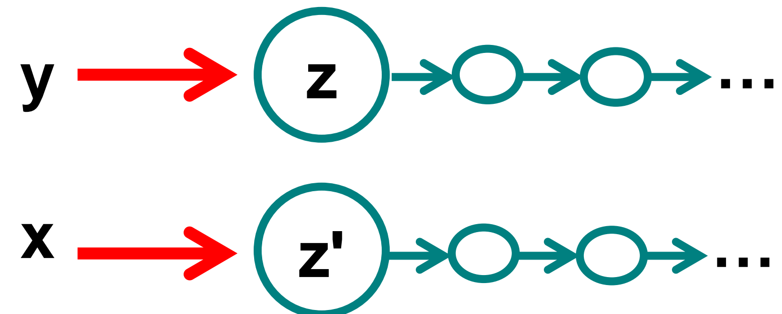
1. Reference copy
   - Makes copy of reference
   - x = y;



2. Shallow copy
   - Makes copy of object
   - x = y.clone( );



3. Deep copy
   - Makes copy of object z and all objects (directly or indirectly) referred to by z



Constructor

- Description
  - Method invoked when object is instantiated
  - Helps initialize object
  - Method with same name as class w/o return type
  - Default parameterless constructor
    - If no other constructor specified
    - Initializes all fields to 0 or null
  - Implicitly invokes constructor for superclass
    - If not explicitly included

Constructor – Example

```
class Foo {
    Foo( ) { … }                // constructor for Foo
}
class Bar extends Foo {
   Bar( ) {                     // constructor for Bar
                               // implicitly invokes Foo( ) here
       …
   }
}
class Bar2 extends Foo {
   Bar2( ) {                    // constructor for bar
      super();                  // explicitly invokes Foo( ) here
   }
}
```

Initialization Block

- Definition
  - Block of code used to initialize static & instance variables for class
- Motivation
  - Enable complex initializations for static variables
    - Control flow
    - Exceptions
  - Share code between multiple constructors for same class

Initialization Block Types

- Static initialization block
  - Code executed when class loaded
- Initialization block
  - Code executed when each object created (at beginning of call to constructor)
- Example

```
class Foo {
    static { A = 1; }        // static initialization block
    { A = 2; }               // initialization block
}
```

Variable Initialization

- Variables may be initialized
  - At time of declaration
  - In initialization block
  - In constructor
- Order of initialization
  1. Declaration, initialization block
     (in the same order as in the class definition)
  2. Constructor

Variable Initialization – Example

```
class Foo {
    static { A = 1; }        // static initialization block
    static int A = 2;        // static variable declaration
    static { A = 3; }        // static initialization block
    { B = 4; }               // initialization block
    private int B = 5;       // instance variable declaration
    { B = 6; }               // initialization block
    Foo() {                  // constructor
      A = 7;
      B = 8;
    }                        // now A = 7, B = 8
}          // initializations executed in order of number
```

Garbage Collection

- Concepts
  - All interactions with objects occur through reference variables
  - If no reference to object exists, object becomes garbage (useless, no longer affects program)
- Garbage collection
  - Reclaiming memory used by unreferenced objects
  - Periodically performed by Java
  - Not guaranteed to occur
  - Only needed if running low on memory

Destructor

- Description
  - Method with name finalize()
  - Returns void
  - Contains action performed when object is freed
  - Invoked automatically by garbage collector
    - Not invoked if garbage collection does not occur
  - Usually needed only for non-Java methods
- Example

```
class Foo {
    void finalize() { … }          // destructor for foo
}
```

Method Overloading

- Description
  - Same name refers to multiple methods
- Sources of overloading
  - Multiple methods with different parameters
    - Constructors frequently overloaded
  - Redefine method in subclass
- Example

```
class Foo {
    Foo( ) { … }          // 1st constructor for Foo
    Foo(int n) { … }      // 2nd constructor for Foo
}
```

Package

- Definition
  - Group related classes under one name
- Helps manage software complexity
  - Separate namespace for each package
    - Package name added in front of actual name
  - Put generic / utility classes in packages
    - Avoid code duplication
- Example

```
package edu.umd.cs;      // name of package
```

Package – Import

- Import
  - Make classes from package available for use
  - Java API
    - java.* (core)
    - javax.* (optional)
- Example

```
import java.util.Random; // import single class
import java.util.*;      // all classes in package
…                        // class definitions
```

Scope

- Scope
  - Part of program where a variable may be referenced
  - Determined by location of variable declaration
    - Boundary usually demarcated by { }
- Example

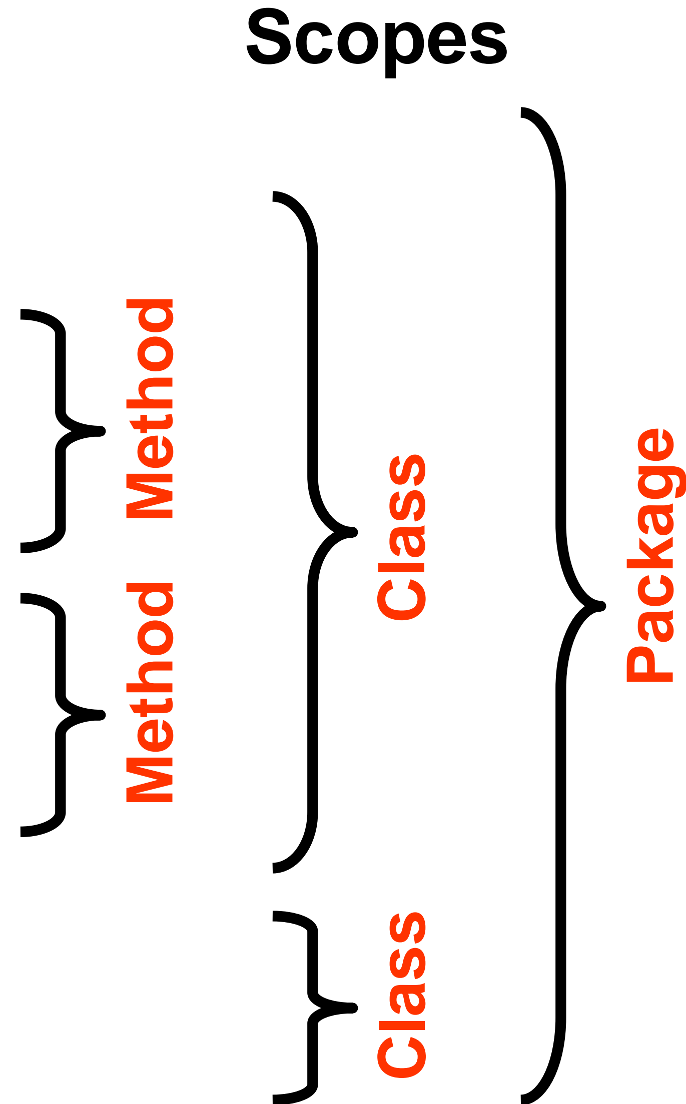
```
public MyMethod1() {
    int myVar;
    ...
}
```

myVar accessible in method between { }

Scope – Example

- Example

```
package edu.umd.cs ;
public class MyClass1 {
    public void MyMethod1() {
        ...
    }
    public void MyMethod2() {
        ...
    }
}
public class MyClass2 {
}
```



Inner Classes

- Description
  - Class defined in scope of another class
- Property
  - Can directly access all variables & methods of enclosing class (including private fields & methods)
- Example

```
public class OuterClass {
    private Object value;
    public class InnerClass {
        ...Object x = value;
    }
}
```

Modifier

- Description
  - Java keyword (added to definition)
  - Specifies characteristics of a language construct
- (Partial) list of modifiers
  - Public / private / protected
  - Static
  - Final
  - Abstract

Modifier

- Examples

```
public class Foo {
    private static int count;
    private final int increment = 5;
    protected void finalize { … }
}
public abstract class Bar {
    abstract int go( ) { … }
}
```

Visibility Modifier

- Properties
  - Controls access to class members
  - Applied to instance variables & methods
- Four types of access in Java
  - Public
  - Protected
  - Package
    - Default if no modifier specified
  - Private

Most visible

Least visible

Visibility Modifier – Where Visible

- “public”
  - Referenced anywhere (i.e., outside package)
- “protected”
  - Referenced within package, or by subclasses outside package
- None specified (package)
  - Referenced only within package
- “private”
  - Referenced only within class definition
  - Applicable to class fields & methods

Visibility Modifier

- For instance variables
  - Should usually be private to enforce encapsulation
  - Sometimes may be protected for subclass access

- For methods
  - Public methods – provide services to clients
  - Private methods – provide support other methods
  - Protected methods – provide support for subclass

Modifier – Static

- Static variable
  - Single copy for class
  - Shared among all objects of class
- Static method
  - Can be invoked through class name
  - Does not need to be invoked through object
  - Can be used even if no objects of class exist
  - Can not reference instance variables

Modifier – Final

- **Final variable**
  - Value can not be changed
  - Must be initialized in every constructor
  - Attempts to modify final are caught at compile time
- **Final static variable**
  - Used for constants
  - Example

```
final static int Increment = 5;
```

Modifier – Final

- Final method
  - Method can not be overridden by subclass
  - Private methods are implicitly final
- Final class
  - Class can not be a superclass (extended)
  - Methods in final class are implicitly final

Modifier – Final

- Using final classes
  - Prevents inheritance / polymorphism
  - May be useful for
    - Security
    - Object oriented design
- Example – class String is final
  - Programs can depend on properties specified in Java library API
  - Prevents subclass that may bypass security restrictions

Modifier – Abstract

- Description
  - Represents generic concept
  - Just a placeholder
  - Leave lower-level details to subclass
- Applied to
  - Methods
  - Classes
- Example

```
abstract class Foo {                // abstract class
    abstract void bar( ) { … }      // abstract method
```

Abstract – Motivating Example

- **Graphics drawing program**
  - **Define a base class Shape**
  - **Derive various subclasses for specific shapes**
  - **Each subclass defines its own method drawMe( )**

```
public class Shape {
    public void drawMe( ) { … }        // generic drawing method
}
public class Circle extends Shape {
    public void drawMe( ) { … }        // draws a Circle
```

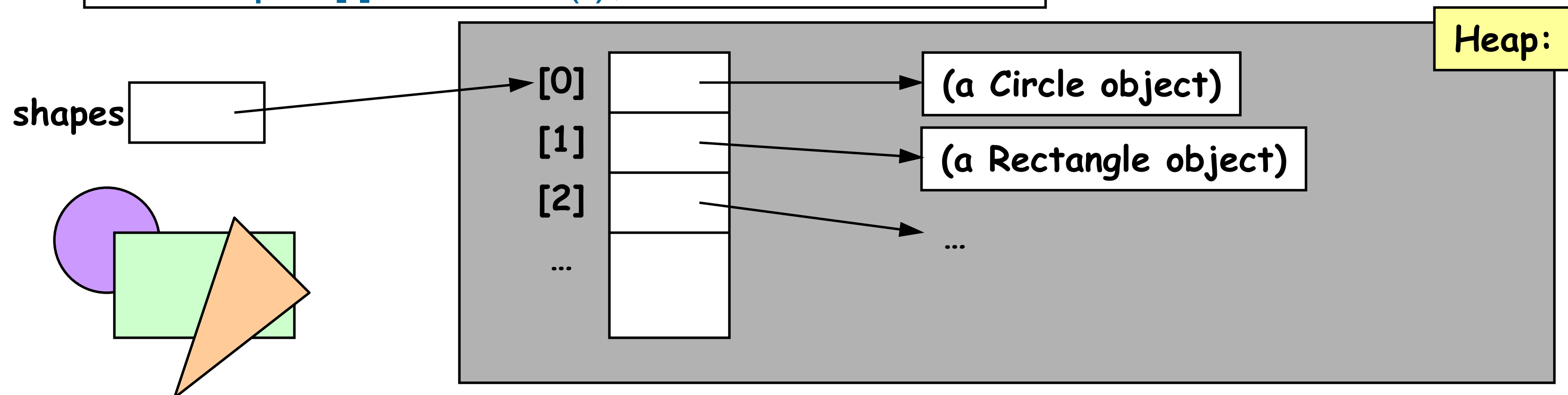
Motivating Example – Shapes

- **Implementation**
  - **Picture consists of array shapes of type Shape[ ]**
  - **To draw the picture, invoke drawMe( ) for all shapes**

```
Shape[ ] shapes = new Shape[…];
shapes[0] = new Circle( … );
shapes[1] = new Rectangle( … );
…
for ( int i = 0; i < shapes.length; i++ )
    shapes[i].drawMe( );
```

Store the shapes to be drawn in an array.

Draws all the shapes. Each call invokes drawMe for the specific shape.

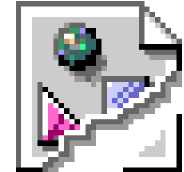


Motivating Example – Shapes

- Problem
  - Shape object does not represent a specific shape
    - Since Shape is just a superclass
  - How to implement Shape's drawMe( ) method?

```
public class Shape {
   void drawMe( ) { … }   // generic drawing method
}
```

Motivating Example – Shapes

- Possible solutions
  - Draw some special "undefined shape" 
  - Ignore the operation
  - Issue an error message
  - Throw an exception

- Better solution
  - Abstract drawMe( ) method, abstract Shape class
  - Tells compiler Shape is incomplete class

Abstract Method

- Behaves much like method in interface
- Give a signature, but no body
- Includes modifier abstract in method signature
- Class descendents provide the implementation
- Abstract methods cannot be final
  - Since must be overridden by descendent class
  - Final would prevent this

Abstract Class

- **Required if class contains any abstract method**
- **Includes modifier abstract in the class heading**

```
public abstract class Shape { … }
```

- **An abstract class is incomplete**
  - **Cannot be created using “new”**

    ```
    Shape s = new Shape( … );          // Illegal!
    ```

  - **But can create concrete shapes (Circle, Rectangle) and assign them to variables of type Shape**

    ```
    Shape s = new Circle( … );
    ```

Example Solution – Shapes

```
public abstract class Shape {
    private int color;
    Shape ( int c ) { color = c; }
    public abstract void drawMe( );
}
```

Base class **Shape** is abstract because it contains the abstract (undefined) method drawMe( ).

```
public class Circle extends Shape {
    private double radius;
    public Circle( int c, double r ) { … details omitted … }
    public void drawMe( ) { … Circle drawing code goes here …  }
}
```

Derived class **Circle** is concrete because it defines drawMe( ).

```
public class Rectangle extends Shape {
    private double height;
    private double width;
    public Rectangle( int c, double h, double w ) { … details omitted … }
    public void drawMe( ) { … Rectangle drawing code goes here …  }
}
```

Derived class **Rectangle** is concrete because it defines drawMe( ).

The code for drawing the shapes given earlier can now be applied.

Abstract – Summary

- Abstract methods
  - Method that contains no body
  - Subclass provides actual implementation

- Abstract classes
  - Required if any method in class is abstract
  - Can contain non-abstract methods
  - Can be partial description of class

Generic Programming

- Generic programming
  - Defining constructs that can be used with different data types
  - I.e., using same code for different data types

- Implemented in Java through
  1. Inheritance → A extends B
  2. Type variables → <A>

Generic Programming Examples

- **Inheritance**

```
Class A {
   doWork( A x ) { … }
}
Class B extends A { … }

A w1 = new A( );
B w2 = new B( );

w1.doWork( w1 );
w2.doWork( w2 );
```

- **Type Variables**

```
Class W<T> {
   doWork( T x ) { … }
}
Class A { … }
Class B { … }

W<A> x1 = new W<A>( );
W<B> x2 = new W<B>( );
A w1 = new A( );
B w2 = new B( );

x1.doWork( w1 );
x2.doWork( w2 );
```

doWork( ) applied to objects of both class A and B

Generic Class

- Class with one or more type variables
  - Example → class ArrayList<E>
- To use generic class, provide an actual type
  - Valid types
    - Class → ArrayList<String>
    - Interface → ArrayList<Comparable>
  - Invalid types
    - Primitive type → ArrayList<int>

      (use wrappers) → ArrayList<Integer>

Defining a Generic Class

- Append type variable(s) to class name
  - Use angle brackets → ClassName<type variable>
- Can use any name for type variable
  - But typically single uppercase letter → T, E, etc…
- Use the type variable to define
  - Type of variables
  - Type of method parameters
  - Method return type
  - Object allocation

Example Generic Class

- Example

```
public class myGeneric<T> {
    private T value;
    public myGeneric( T v ) { value = v; }
    public T getVal( ) { return value; }
    public void setVal( T newV ) { value = newV; }
}
```