Introduction to Object-Oriented Programming
Part II: OOP Concepts

Course by Mr. Erkki Mattila, M.Sc.
Rovaniemi University of Applied Sciences

Course Contents

- Part I: Object-Oriented Concepts and Principles
- Part II: Object-oriented Programming
  - Closer look at the concepts of OOP

Member Variables

- Variables defined inside a class are called member variables
- Member variables fall into two categories: instance variables and class variables (static variables)

Instance Variables

- All objects have their own copies of instance variables
- System allocates memory for instance variables at object creation time
- Instance variables can be accessed only through an instance, for example
  Vector v = new Vector(); int c=v.elementCount;
- The state of an object is specified by the values of its instance variables

Class (Static) Variables

- The runtime system allocates a class variable once per class, regardless of the number of instances created of that class
  - All instances of the same class share the same copy of class variables
- The system allocates memory for a class variable the first time it encounters the class (=loads it into memory)
- You can access class variables either through an instance or through the class itself
  - The latter is better style, for example String s = Integer.toString(10);

Example of Instance and Class Variables in Java

```
public abstract class Shape
{
   public static int a_numberOfShapes=0;

   private Color a_color;

   public Shape(){
      Shape.a_numberOfShapes++;
      a_color = Color.BLACK;
   }

   public Shape(Color c){
      Shape.a_numberOfShapes++;
      a_color = c;
   }
}
```

Methods

- The code that describes how to perform an operation on a specific object type is called a method
- In different OOP languages, methods can also be called as functions or operations, all meaning the same thing.
- Variables and methods can be called with a common name class members
- Methods fall into two categories: instance methods and class methods

Instance Methods

- Both instance and class members can be referenced from an instance method
- If no instance name is given when referring to an instance variable or method, the system uses the current instance (object self-reference)

Class (Static) Methods

- A class method is a static method in some class.
- It may not access the instance variables or methods of the class, but only the other static members of the class
- A static method can access an instance variable or method only if the instance (object) name is mentioned before the variable/method name

Method Overloading

- Method overloading means having two methods with the same name **in the same class** that differ in the amount and/or types of their arguments.
- It is not legal to have two methods of the same name that differ **only** in their return types
- Method overloading is not an O-O feature; it can exist in non-object languages as well.
  - Do not mix with method overriding!

Example of Method Overloading

```
public class Rectangle extends Shape
{
   private int width;
   private int height;

   public Rectangle() {
      width=0;
      height=0;
   }
   public Rectangle(int w, int h) {
      width=w;
      height=h;
   }
   public Rectangle(int w, int h, Color c) {
      super(c);
      width=w;
      height=h;
   }
}
```

Coercion

- Coercion and overloading often go hand in hand
- Coercion occurs when an argument of one type is converted into the expected type automatically
- Example:

```
public float add( float a, float b );
int x=1, y=2;
float z = add(x,y);
```

  - When you call add() with int arguments, the arguments are converted into floats by the compiler (type cast)
  - To avoid coersion another overloaded method with int type of arguments could be added to the class:

Exercise

- Look through the Java API. Find an example of overloading and explain it.

Variable Shadowing

```
public class Circle extends Shape
{
   private int radius;

   // Parameter radius shadows member variable radius
   public Circle(int radius) {
      this.radius=radius;
   }

   public Circle(int r, Color circleColor) {
      super(circleColor);
      radius=r;
   }
}
```

- Shadowing means using the same names for member variables **and** method parameters/local variables
- Sometimes used, but better to be avoided

Naming the Member Variables

- A common way to avoid variable shadowing is to follow a naming convention for member variables
- For instance add a prefix to each member variable:
  - m_ as member, or
  - a_ as attribute
  - The latter one might be better, because letter m can also be incorrectly associated with the word method
  - Example: private int a_size;
- For local variables and parameters it is then possible to use the same name without the prefix
- How could you fix the example on the previous slide to avoid variable shadowing?

Variable Shadowing Example Fixed

- Rename member variable radius to a_radius
- Parameter radius no longer shadows the member variable

```
public class Circle extends Shape
{
    private int a_radius;

    public Circle(int radius) {
        a_radius=radius;
    }
}
```

Object Self-reference

- In OOP languages, a special keyword is used to refer to the current instance of an object
  - *this* in Java, C++ and C#
- Self-reference contains the object type and the value of the current object instance
- Self-reference is used to access the object whenever a reference to it is needed inside the object
  - to access a member variable when shadowed by local variables
  - to pass the object itself as a function argument
  - to add the object itself to a collection
  - to call another constructor method in the

Constructor

- A specialized method used to instantiate an object
- The constructor function has the same name as the class. It is never called directly, but the run-time system calls it when a new object is created
- Constructors are usually overloaded; a class contains multiple constructors, which have a different set of parameters

Destructor

- Destructor is a specialized method, which is called when an object is deleted
- Its purpose is to release the resources the object may have allocated
- Destructor cannot be overloaded and it cannot have any arguments
- Constructor and destructor methods can't be inherited
  - When a class is instantiated, the system first calls the constructor of its superclass and the superclass's superclass until it reaches the root of the inheritance hierarchy

Constructor Example 1

```
public class Rectangle extends Shape
{
   private int width;
   private int height;

   public Rectangle() {
      width=0;
      height=0;
   }

   public Rectangle(int w, int h)
   {
      width=w;
      height=h;
   }

   public Rectangle(int w, int h, Color c)
   {
      super(c);
      width=w;
      height=h;
   }
}
```

Constructor Example 2

```
public class TestShape
{
  public void someMethod()
  {
    Color myColor = new Color(100, 170, 15);

    // Call parameterless consructor
    Rectangle r1 = new Rectangle ();

    // Call constructor with two int type of parameters
    Rectangle r2 = new Rectangle(5,8);

    // Call constructor with two int and one Color type of parameters
    Rectangle r3 = new Rectangle(7, 15, myColor);
  }
}
```

Method overriding

- Method overriding means providing a replacement method in a new class for a method inherited from the base class; i.e. we give a new implementation for an inherited superclass method in the subclass
- The replacement method must have exactly the same name, argument list and return type as the inherited method
- Overriding occurs when attributes and methods are inherited in the normal manner, but are then modified to the specific needs of the new class

Example of Method Overriding 1

```
public abstract class Shape
{
   public static int a_numberOfShapes=0;
   protected Color a_color;

   public Shape(Color c) {
      Shape.a_numberOfShapes++;
      a_color = c;
   }

   public long getArea() { return 0; };

   public abstract long getCircumference();

}
```

Example of Method Overriding 2

```
public class Circle extends Shape
{
   private int a_radius;

   public Circle(int radius, Color circleColor) {
      super(circleColor);
      a_radius=radius;
   }
   public long getCircumference() {
     return Math.round(2* Math.PI*m_radius);
   }
   public long getArea() {
     return Math.round(Math.PI*m_radius*m_radius);
   }
}
```

Redefinition

- Occurs when a subclass defines a method **using the same name** as a method in the superclass **but with a different type signature**
- **The change in type signature (parameter list) is what differentiates redefinition from overriding**
- Two different techniques to resolve the redefined name: merge model (Java, C#) and hierarchical model (C++)
- Merge model: all the currently active scopes are examined to find the best match
- Hierarchical model: scopes are examined one by one, starting from the subclass scope. If the name is defined there, the closest match in that scope will be the one selected
- Redefinition is bad programming style: easy to cause confusion and errors

Example of Redefinition

```
class Parent {
  public void example(int a) {…}
}
class Child extends Parent {
  public void example(int a, int b) {}
```

- Let's create an instance of the child and execute the method with one argument:

```
Child c = new Child();
c.example(7);
```

- In Java and C#, the method from the parent class will be selected
- In C++ the code will produce a compilation error. Why?

Variable Hiding

- Within a class, a member variable that has the same name as a member variable in the superclass hides the superclass's member variable
- Within the subclass, the member variable in the superclass can no longer be referenced by its simple name
  - Instead, the superclass member variable must be accessed through the **super** operator (Java, C++)
- Variable hiding is bad programming style
  - Easy to cause confusion and errors
- Just because a programming language allows you to do something, it does not always mean that it is a desirable thing to do!

Example of Variable Hiding

```
class Employee {
   protected int salary;
   protected int hours;
}
class PartTimeEmployee extends Employee {
   protected int salary;
   public void setSalary(int sal) {
     super.salary = sal; // Sets the superclass salary
   }
   public int calculateMonthlySalary() {
     // Uses superclass hours, but subclass salary
     return hours * salary;
   }
}
```

Exercise

- Consider the Java API class java.io.Writer
- What kind of examples of method overriding can you find in the subclasses of class Writer?
- Find another example of method overriding in the Java API

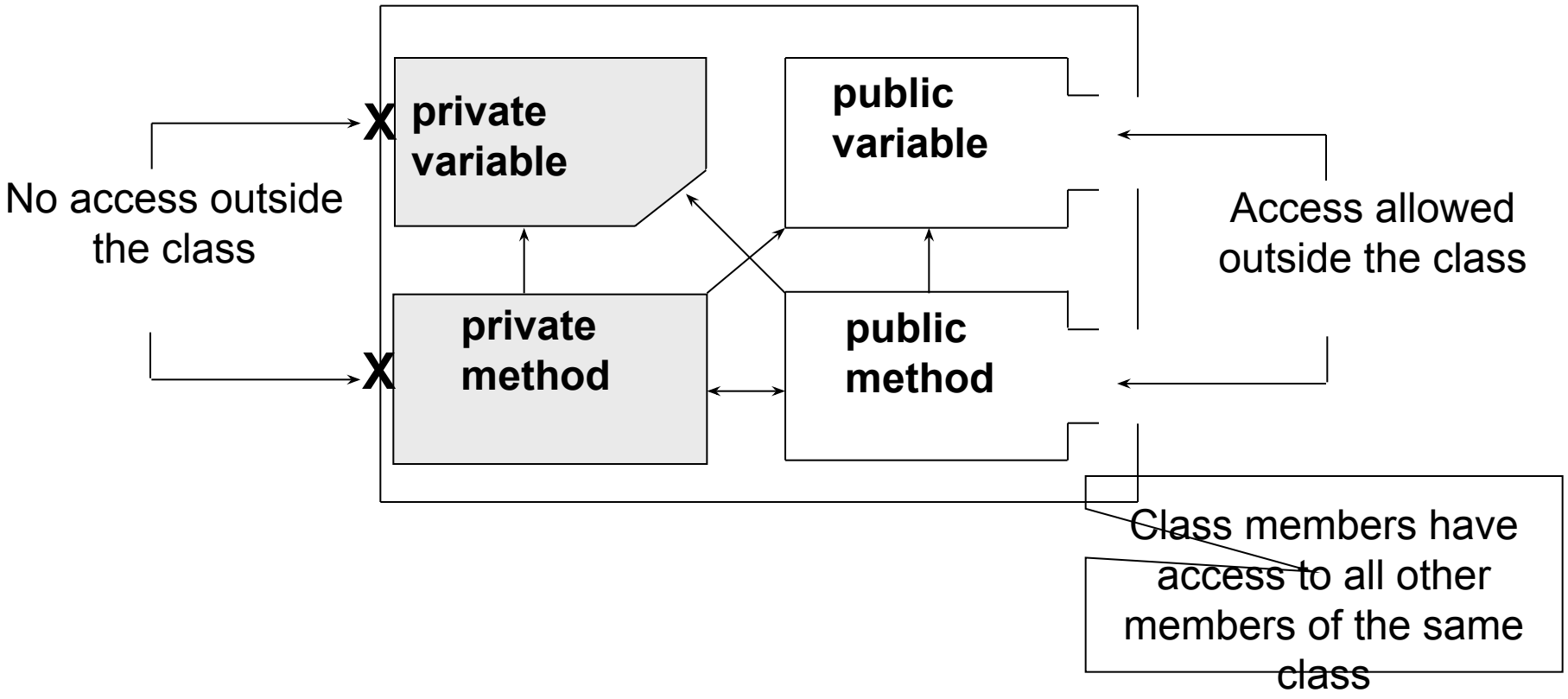
Access Specifiers (Visibility Identifiers)

- There are three parts in a class, which have different protection level:
  - public part
  - protected part
  - private part
- An access specier is given to a class and to all members of the class
- Public variables and methods are visible to all classes
- Protected members are only visible inside the class itself and its subclasses

Access Specifiers

- Private members are only visible inside the class itself
- Public and protected parts define the class's interface to other classes
- Constructor methods must be declared as public
  - Otherwise the class cannot be instantiated as constructors are always called outside the class itself
  - With the rare exception of singleton pattern

Access Specifiers



Private Data – Public Accessor methods

- Member variables are usually declared as private
- Public get and set methods are used to set and get the variable values
- Advantages of using set and get methods:
  - We can give a read-only access to certain data by specifying only the get method
  - We can check for the validity of the data in the set method, e.g. not allow negative values for person's age or height
  - We can ensure that required updates are done, e.g. a field value is updated on the screen as well
  - We can change the data structures inside the class, yet keep its public interface untouched causing no side-effects outside the class

Checking Parameters for Validity

- Methods should be made as general as practical
  - Do not set arbitrary restrictions on method parameters
- Yet most methods have restrictions on what values may be passed into their parameters
  - E.g. no negative values, no null values
- Document clearly all such restrictions
- Check the validity of parameters before the execution of the method
- If the check fails, exit quickly and cleanly with an appropriate exception
  - Public methods should throw an appropriate exception (Java: Null Pointer-, IllegalArgument-, IndexOutOfBounds-, etc.) For non-public methods use assertions, since you are the author of the package and therefore responsible for any illegal method calls

Example of Using Access Speciers with Set and Get Methods

```
public class Circle extends Shape
{
   private int radius;
   public Circle() {
      radius=0;
   }
   public int getRadius() {
      return radius;
   }
   public void setRadius(int r) {
      if (r >= 0)     // Checking validity
        radius = r;
      else
        radius = 0;
   }
}
```

Exercise: Access Specifiers - Set and Get Methods

- Modify the class Shape so that the colour will be represented with three int type of variables red, green and blue instead of a single Color type of a variable. Red, green and blue are integer values from 0 to 255.
- Make the required mapping between the new and old data structure in the setColor, getColor and constructor –methods of class Shape.
  - Only change the method implementations, not the prototypes/signatures
  - You can create a new Color type of an object by giving the integer values as parameters to Color class constructor
    For instance Color newColor = new Color(r, g, b);
  - You can get the RGB –values from a Color type of an object using the methods getRed(), getGreen() and getBlue()
    For instance int green = newColor.getGreen();

Exercise: Checking Parameter Values for Validity

- Add set and get methods for the new int type of member variables **red**, **green** and **blue** in the modified Shape class
- Check in the set methods that the parameter value is in the range from 0 to 255. If it is not, leave the attribute value unchanged (and throw an exception or return an error code)

Using `Protected` Instance Variables

- Advantages
  - Subclasses can modify values directly
  - Slight increase in performance
    - Avoid set/get function call overhead
- Disadvantages
  - No validity checking
    - subclass can assign illegal value
  - Implementation dependent
    - subclass methods more likely dependent on superclass implementation
    - superclass implementation changes may result in subclass modifications
      - Fragile (brittle) software

Abstract Class

- Abstract methods are methods that are declared but not yet implemented
  - Abstract methods have name, return value and arguments, but no method body (implementation)
  - Also called as deferred methods, or in C++ pure virtual methods
- Only abstract classes may have abstract methods; if a class has even one abstract method the class itself must be declared as abstract
- An abstract class cannot be instantiated, but it can be inherited

Purpose of Abstract Classes

- Allow a programmer to define common interface for a group of classes
- Allow a programmer to use polymorphic method calls (late binding)
  - For example in a collection of classes representing geometric shapes, we can define a method to calculate the area of the shape in each of the subclasses Circle, Rectangle, Triangle
  - Suppose a programmer defines a polymorphic variable of class Shape that will, at various times, contain instances of each different shape. Compiler will permit message **getArea()** to be used with this variable only if it can ensure that the message will be understood by any value that can be associated with the variable
    - Solution is to declare an abstract method **getArea()** in the superclass Shape

Abstract Class vs. Interface

- Besides abstract methods, an abstract class can contain variable definitions and method implementations
  - Can inherit other abstract classes and implement interfaces
- An interface can only contain abstract methods and constant definitions
  - ”Pure abstract class”
  - Can inherit other interfaces and only them
- Purpose of Interfaces
  - Interfaces define and standardize the ways in which people and systems can interact with one another

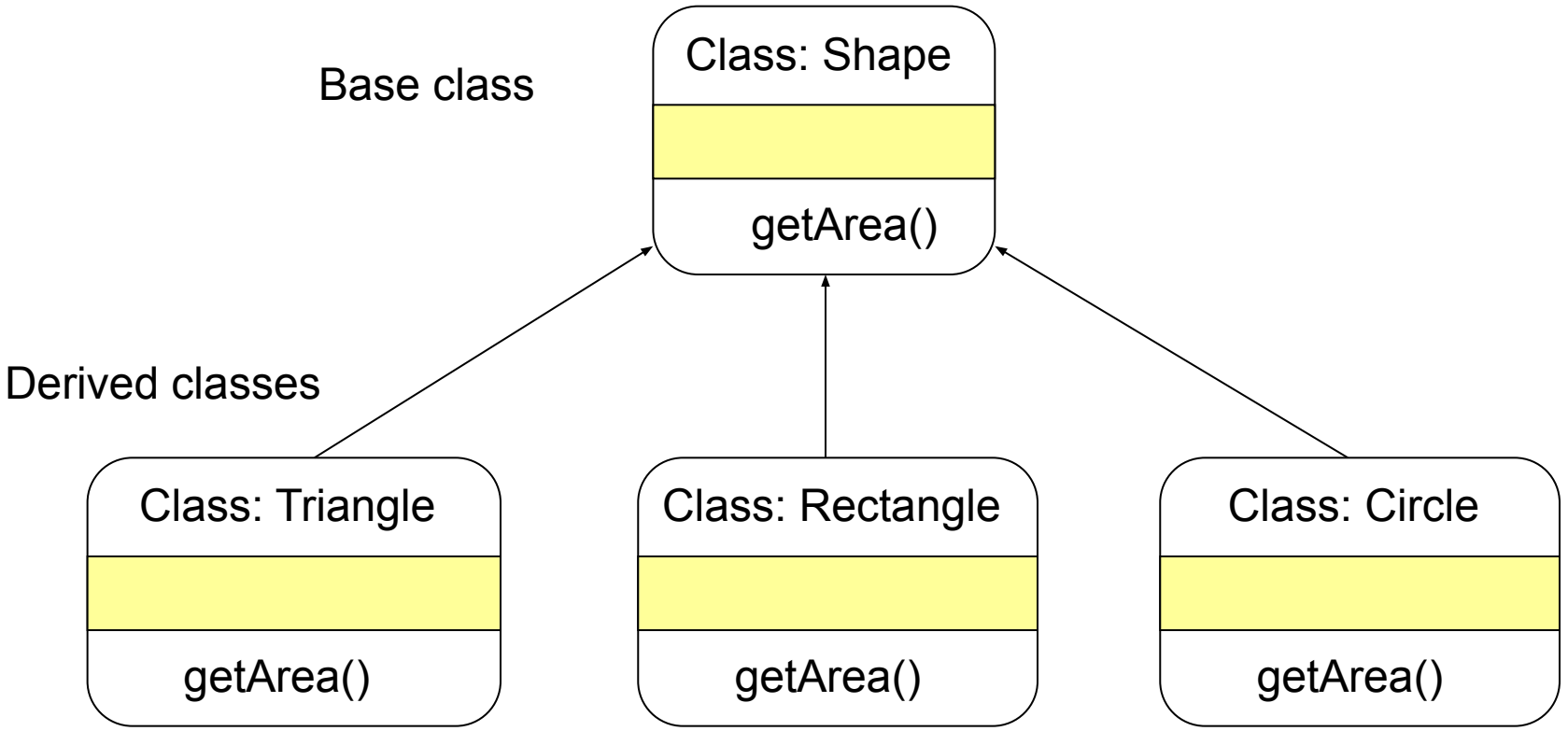
Polymorphism

- Generally, the ability to appear in many forms
- In OOP polymorphism refers to a programming language's ability to process objects differently depending on their class
- Polymorphism is a characteristic that greatly reduces the effort required to extend an existing OO system

Polymorphism in OOP

1. We can refer to a subclass type of an object with a superclass type of a variable
   - The class of the referred object is not known at compile time
   - Responds at run time according to the actual class of the referred object (late binding)
2. Subclasses can override inherited superclass methods; we can have several implementations of the same method in the class hierarchy
   - For example, given a base class Shape, polymorphism enables the programmer to define different getArea() method for any number of derived classes, such as Circle, Rectangle and Triangle
   - No matter what shape an object is, applying the

Polymorphism Example 1



Polymorphism Example 2

- A simple example of polymorphism is the method append in the Java class StringBuffer
- The argument to this method is declared as **Object** and thus can be any object type
- The method has the following definition:

```
class StringBuffer {
  String append(object value) {
    return append( value.toString() );
  }
}
```

- The method **toString** is defferred. It is defined in class Object and redefined in a large number of different subclasses
- Each of these definitions of toString will have slightly different effect: a Double will produce a textual representation of a numeric value; Color will generate a string that describes the red, green and blue values in the color, etc.

Exercises

1. In your own words, explain polymorphism
2. Consider again the Java API class java.io.Writer.
   - (What kind of examples of method overriding can you find in the subclasses of class Writer?)
   - When programming, you should write your objects and methods to act on instances of Writer (instead of subclass types). Why?

Late Binding 1

- Dynamic binding of messages (method calls) to method definitions
- Polymorphism ensures that the proper version of the method is called based on the type of the object
- Late binding only applies to instance methods; this sort of dynamic lookup does not happen for static/class methods. Why?

Late Binding 2

- Actual operations can be bound to messages as soon as the type of the object receiving the message is known
  - If this is known during the compilation, an early binding occurs, otherwise the mapping is done on run-time, late binding occurs
- Early binding is typically more efficient in terms of hardware resource usage
- Late binding is a mechanism to implement polymorphism, which is one of the main principles of OOP
  - Reminder: polymorphism - different type of objects invoke different methods in response to the same message (method call)

Virtual Operation

- An operation which late binding can be applied to is called a virtual operation
- In Java, all methods are virtual by default Method overriding can be prohibited by defining the method as *final*
- In C++ all methods are non-virtual by default. Methods must by specified as virtual by using the *virtual* keyword

Group Work

- Write down the names of the Shape, Triangle, Rectangle and Circle classes (see previos slide)
- What kind of attributes the classes should have (to be able to calculate the area of the shape they represent)?
- Find common attributes, if any, and move them to the common base class Shape
- Add an abstract method getArea() to the base class
- Implement the getArea() method in each of the subclasses. It should return the area of that particular shape
- Area of a triangle = ½absinC, Area of a circle = ¶r^2
- Similarly add method getCircumference() to the superclass and subclasses

Group Work

- Add a new subclass for the class Shape, for instance Parallelogram (area = side1 * side2 * sin(angle))
  - Use the functions Math.sin(angle in radians) and Math.toRadians(angle in degrees)
- Create an instance of the new class in the Test class and call the printAreaAndCircumference for it

Other Forms of Polymorphism

- The type of polymorphism described earlier is called pure polymorphism or inclusion polymorphism
  - Pure polymorphism occurs when a polymorphic variable is used as an argument in a method
- Another type of polymorphism is parametric polymorphism, which is implemented in Java as generic classes and in C++ with template classes
- Method overriding and overloading can also be considered as types of polymorphism

Multiple Inheritance

- Sometimes it is tempting to inherit some attributes and methods from one class and others from another class. This is called multiple inheritance
- Multiple inheritance means having more than one direct superclass
- Multiple inheritance complicates the class hierarchy and creates potential problems in configuration control (sequences of multiple inheritance are more difficult to trace)

Problems with Multiple Inheritance

- Name ambiguity
  - Inherited, different features can have the same name
  - Same feature may be inherited several times
- Impact on substitutability
  - Overriding a method that has been inherited from several superclasses
  - deck.draw(), rectangle.draw()
- Many types of tricks invented to overcome the problems
  - Make the program complicated, is it worth it?

Reference and Value Semantics

- Reference semantics: variable values are references to objects (Java)
  - Assignment x=y causes a pointer copy: both x and y refer to the same object after the assignment has been done.
- Value semantics: variable values are the objects (default in C++)
  - Assignment x=y causes all the field values of object y to be copied to the fields of object x. Two separate, identical objects exist after the assigment has been performed

Reference and Value Semantics

- Pure object languages use reference semantics
  - Value semantic type of copying can be done with a separate cloning method.
- Hybrid languages (such as C++) contain a special type for handling object references: a pointer type
  - Assignment xPointer = yPointer causes a pointer copy: both xPointer and yPointer refer to the same object after the assigment has been done.

Copying Objects

- Pointer copy
  - According to reference semantics, creates a new reference to the same object
  - Occurs when assignment operator is applied to pointers in C++, or to object references in Java
- Shallow copy
  - Creates a separate copy of the object
  - Pointer or reference members refer to the same objects as in the original object
  - Default behaviour in C++ when assignment operator is applied to object values
  - In Java the default clone method produces a shallow copy
- Deep copy
  - Creates a separate copy of the object
  - The objects referred to by pointer or reference members are also copied
  - In C++ you must define a copy constructor and overload the assignment operator to enable creating a deep copy of an object
  - In Java serialization can be used to create a deep copy of an object