# CS571: Programming Languages



## Object-Oriented Programming

- Based on a notion of an object: a collection of memory locations together with all the operations that can change the values of these memory locations.
- Began in the 1960s with the simula project
  - To corporate into the language the notion of an object, which is similar to a real-world object.



## Software Reuse

- Basic ways that a software component can be modified for reuse:
- Extension of the data and/or operations.
  - Example:

    A window is defined on a computer screen as a rectangle specified by its four corners, with operations that may include resize, display and erase.

    A text window can be defined as a window with some added text to be displayed.



## Software Reuse (Cont.)

- Restriction of the data and/or operations
  - Example: A rectangle has both a length and a width, but a square has length equal to width, so one piece of data can be dropped in creating a square.
  - Rarely seen in programming languages, but it does occasionally arise in practice as a contrast to extension.
- Redefinition of one or more of the operations
  - Example: If a square is obtained from a rectangle and an area or perimeter function may need to be redefined to take into account the reduced data needed in the computation.



## Software Reuse (Cont.)

- Collection of similar operations from two different components into a new component.
  - Example

    A circle and a rectangle are both objects that have position and that can be displayed.

    These properties can be combined into an abstract object called a figure.



## Software Reuse (Cont.)

- Polymorphization: the extension of the type of data that operations can apply to
  - Parametric polymorphism: ability of a function to take arguments of multiple types.
  - Overloading: same function name is used to represent different functions, each of which may take arguments of a specific type.

## Encapsulation and Information Hiding

- Encapsulation: enables the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users.
- Information hiding: separating interface from implementation:
  - We can replace the implementation of an object easily, without making any changes to client code
- These two terms overlap to some extend.



## Class and Objects



## Class and Object

- Class is a type and includes data and operations together.
- An object is an instance of class.
  - Objects are declared to be of a particular class exactly as variables are declared to be of a particular type in C.



## Variables

- Member variables
  - Instance variables:
  - Class (static) variables:



## Variables

- Member variables
  - Instance variables:
    - each object has its own copy of instance variables
    - One copy per instance
    - Must be accessed through an object
  - Class (static) variables:
    - across all objects of a class
    - similar to global variables
    - One copy for the class



## Methods

- Each object includes a set of functions called methods.
  - Similar to ordinary procedures and functions
  - Can automatically access and change the object's data.
  - Calling/invoking a method of an object is sometimes called sending the object a message.

## Access Level (Java)



## Access Level (Java)

- By default, a class can be used only by instances of other classes in the same package (package-private).
- A class can be declared public to make it accessible to all class instances regardless of what package its class is in.

```
public class Point
{ …}
```



## Access Level: Fields and Methods (Java)



## Access Level: Fields and Methods (Java)

- Fields and methods can be declared package, public, private, or protected.
  - package: accessible to other classes in the same package.
  - public: accessible to any class in any package.
  - private: accessible only to the class in which it is defined.
  - protected: accessible to the class itself, its subclasses, and classes in the same package.
- If no access level is specified, the field or method access level is package by default.



## Example: Complex Number

- A complex number is a number of the form a+bi
  - a and b are real numbers
  - a: the real part of the complex number
  - b: the imaginary part of the complex number
  - (a1+b1i) + (a2+b2i) =
  - (a1 + b1i)(a2 + b2i) =



## Example: Complex Number

- A complex number is a number of the form a+bi
  - a and b are real numbers
  - a: the real part of the complex number
  - b: the imaginary part of the complex number
  - (a1+b1i) + (a2+b2i) = (a1+a2) + (b1+b2)i
  - (a1 + b1i)(a2 + b2i) =

## Example: Complex Number

- A complex number is a number of the form a+bi
  - a and b are real numbers
  - a: the real part of the complex number
  - b: the imaginary part of the complex number
  - (a1+b1i) + (a2+b2i) = (a1+a2) + (b1+b2)i
  - (a1 + b1i)(a2 + b2i) = (a1a2 – b1b2) + (a1b2 + b1a2)i.



## Example: Complex Number (Java)

```
public class Complex
{    private double re, im;
     public Complex()
     { re = 0; im = 0;}
     public Complex (double realpart, double imagpart)
     { re = realpart; im = imagpart; }
     public double realpart()
     { return re;}
     public double imaginarypart()
     { return im;}
     public Complex add (Complex c) {

     }
     public Complex multiply (Complex c) {

     }
}
```



## Example: Complex Number (Java)

```
public class Complex
{    private double re, im;                    //instance variables
     public Complex()                  //constructor, is called automatically when a
     { re = 0; im = 0;}                // new object of this class is created
     public Complex (double realpart, double imagpart) //constructor
     { re = realpart; im = imagpart; }
     public double realpart()
     { return re;}
     public double imaginarypart()
     { return im;}
     public Complex add (Complex c)
     // all objects must be created with new
     { return new Complex(re + c.realpart(), im + c.imaginarypart());}
     public Complex multiply (Complex c)
     { return new Complex(re * c.realpart() - im * c.imaginarypart(),
                          re * c.imaginarypart() + im * c.realpart()); }
}
```



## The main Method (Java)

- A class can have a main method like this:

```
public static void main(String[] args) {
  …
}
```

- This will be used as the starting point when the class is run as an application
- Keyword static makes this a class method
  - Main method can be called without creating an instance of the class.



## Example: Complex Number (Complex.java)

```
import java.io.*;
public class Main{
     public static void main(String[] args){
          Complex z, w;
          z = new Complex (1,2);     // z.re = 1.0, z.im = 2.0
          w = new Complex (-1,1);   // w.re = -1.0, w.im = 1.0
          z = z.add(w);     // add w to z, create a new Complex object
                            // and assign the new object to z (throwing
                            // away the object previously pointed to by z)
          System.out.println(z.realpart());
          System.out.println(z.imaginarypart());
          z = z.add(w).multiply(z);
          System.out.println(z.realpart());
          System.out.println(z.imaginarypart());
     }
}
```



## Example: Complex Number (Complex.java)

```
import java.io.*;
public class Main{
     public static void main(String[] args){
          Complex z, w;
          z = new Complex (1,2);     // z.re = 1.0, z.im = 2.0
          w = new Complex (-1,1);   // w.re = -1.0, w.im = 1.0
          z = z.add(w);     // add w to z, create a new Complex object
                            // and assign the new object to z (throwing
                            // away the object previously pointed to by z)
          System.out.println(z.realpart());      //0.0
          System.out.println(z.imaginarypart());      //3.0
          z = z.add(w).multiply(z);
          System.out.println(z.realpart());
          System.out.println(z.imaginarypart());
     }
}
```

## Example: Complex Number (Complex.java)

```
import java.io.*;
public class Main{
    public static void main(String[] args){
        Complex z, w;
        z = new Complex (1,2);     // z.re = 1.0, z.im = 2.0
        w = new Complex (-1,1);    // w.re = -1.0, w.im = 1.0
        z = z.add(w);      // add w to z, create a new Complex object
                           // and assign the new object to z (throwing
                           // away the object previously pointed to by z)
        System.out.println(z.realpart());      //0.0
        System.out.println(z.imaginarypart());      //3.0
        z = z.add(w).multiply(z);      //z = (z+w)*z
        System.out.println(z.realpart());      //-12.0
        System.out.println(z.imaginarypart());      //-3.0
    }
}
```



## Class and Object (C++)

- Instance variables and methods are both called members of a class
  - Instance variables: data members
  - Methods: member functions
- Access to members of an object is regulated in C++ using three keywords:



## Class and Object (C++)

- Instance variables and methods are both called members of a class
  - Instance variables: data members
  - Methods: member functions
- Access to members of an object is regulated in C++ using three keywords:
  - Public: can be called by any piece of code
  - Private: can be access only by member functions associated with the class
  - Protected: allows the member functions of any subclass of a given class to access such members
  - A typical convention in C++ is to make all data members private and to make most member functions public.



## Class in C++

Example:

```
class A
{ public:
        // all public members here
  protected:
        // all protected members here
  private:
        // all private members here
};
```

- Unlike Java, C++ does not have built-in garbage collection, instead, it has destructor (provided by the programmer), which is preceded by ~.
- Destructors are automatically called to deallocate memory and do other cleanup for a class object and its class members when an object is deallocated.



## Example (C++)

- A list that consists of integers

```
class IntList {
  private:
    int elem;          //element of the list
    IntList *next;    // pointer to next element in the list
  public:
    IntList(int first); //constructor
    ~IntList();    //destructor.  It is called automatically when
                   // an object of this class is about to be destroyed.
    void insert(int i);   //insert element i
    int getval();         //return the value of an element
    IntList *getNext();       //return the next element
};
```



## Example (C++)

- The scope resolution operator :: allows member functions to have their implementation given outside the declaration of a class.

```
void IntList :: insert(int i) {
        ……
}
```

## Static Variables

- Static variables: across all objects of a class – similar to global variables
  - One copy for the class
  - Can be accessed without class instances



## Example: Static Variables (static.cpp)

```
#include <iostream>
using namespace std;
class CDummy {
      public: static int n;
              CDummy () { n++; };
              ~CDummy () { n--; };
};
int CDummy::n=0;
int main () {
      CDummy a;
      CDummy b;
      CDummy * c = new CDummy;
      cout << a.n << endl;
      delete c;
      cout << CDummy::n << endl;
return 0; }
```

**Output: 3 2**



## Static Variables (C++)

- Is the following correct?

```
class CDummy {
      public: static int n;
              n = 0;
              CDummy () { n++; };
              ~CDummy () { n--; };
};
```

- In order to initialize a static data-member we must include a formal definition outside the class, in the global scope.

  **e.g. int CDummy::n=0; in the previous example.**



## C++ Namespace

- Allows the explicit introduction of a named scope to avoid name clashes among separately compiled libraries.
  - There are likely to be other libraries available in a programming environment that could include a different implementation of a queue, but with the same name.
  - Namespace can be used to disambiguate such name clashes.



## Example: Queue.h (C++)

```
namespace MyQueue
{  struct Queuerep;
   typedef struct Queuerep * Queue;
   Queue createq(void);
   Queue enqueue(Queue q, void* elem);
   void* frontq(Queue q);
   Queue dequeue(Queue q);
   int emptyq(Queue q);
}
```



## Client Code (C++)

- Three options:
  - After the appropriate #include, the user can simply refer to the MyQueue definition using the qualified name, i.e., with MyQueue:: before each name used from the MyQueue namespace.
    - **E.g. Queue q = MyQueue::createeq();**
  - The user can write a using declaration for each name used from the namespace.
  - One can unqualify all the names in the namespace with a single using namespace declaration.

## Client Code (C++)

```
#include <iostream>
#include "queue.h"
using std::endl;  //allow endl from iostream to be used without qualification
using namespace MyQueue;   // allow all names in MyQueue to be
                           //used without qualification
main()
{   int *x = malloc(sizeof(int));
    int *y = malloc(sizeof(int));
    int *z;
    Queue q = createeq();
    .....
    std::cout << *z << endl;
    ……
}
```



## Inheritance



## Inheritance

- Allows the sharing of data and operations among classes as well as the ability to redefine these operations without modifying existing code.

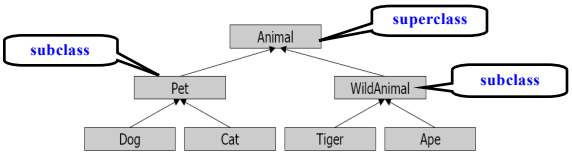


## Inheritance (Cont.)

- A class B can inherit some or all of the instance variables and methods of another class A by declaring A in its definition.
  - B is called a subclass/subtype of A and A a superclass/supertype of B.
    - **Java: public class B extends A**
    - **C++: class B: public/private/protected A**



## Inheritance (Cont.)




- Properties
  - is-a relationship exists between inherited classes. E.g. a pet is an animal, a dog is a pet
  - Inheritance is transitive: inherits from all ancestors.



## Inheritance: Access Level (C++)

- **class B : public A**
  - All public/protected members of A remain public/protected in B. Private members of A are unavailable in B

## Inheritance: Access Level (C++)

- **class B : public A**
  - All public/protected members of A remain public/protected in B. Private members of A are unavailable in B
- **class B : private A**
  - B has private access to the public and protected members of A



## Inheritance: Access Level (C++)

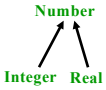
- **class B : public A**
  - All public/protected members of A remain public/protected in B. Private members of A are unavailable in B
- **class B : private A**
  - B has private access to the public and protected members of A
- **class B: protect A**
  - The public and protected parts of A become protected in B.



## Single Inheritance

- Single inheritance: each subclass inherits from only one superclass

tree:






## Example: Single Inheritance (Java)

```
public class A
{ public void f1() {System.out.println("f1");}
}
public class B extends A
{ public void f2() {System.out.println("f2");}
}
```


- B inherits from A. Inheritance graph:
- All of the functions that apply to A also apply to B.

```
B r = new B();
r.f1();
Output: f1
```



## Multiple Inheritance (inherit5.cpp)

- Multiple inheritance: a class may inherit from two or more superclasses
  - C++ provides, Java does not


```
class A {public: void f();};
class B {public: void f();};
class C: public A, B{};
C c;
c.f()          //A's f or B's f?
```

```
error: request for member `f' is ambiguous
error: candidates are: int B::f()
error:                 int A::f()
```



## Multiple Inheritance

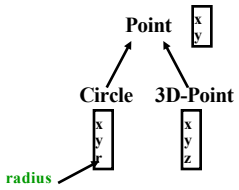
- Multiple inheritance: a class may inherit from two or more superclasses
  - C++ provides, Java does not


```
class A {public: void f();};
class B {public: void f();};
class C: public A, B{};
C c;
c.f()          //A's f or B's f?
c.A::f();
c.B::f();
```

  - Java has multiple interface inheritance which is almost as powerful, and much easier to implement.

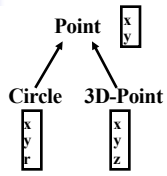
## Two Types of Inheritance

- Extension.
  - A subclass can add data members and member functions of its own.






## Inheritance Example (C++)



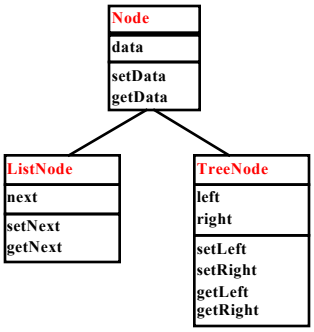

```
class Point{
  protected: int x, y;
  public:
    void set (int a, int b);
};
```

```
class Circle : public Point{
   private: double r;
};
```

```
class 3D-Point: public Point{
   private: int z;
};
```



## Inheritance Example: Node






## Example: Node (Java)

```
public class Node {
   private int data;
   public Node(int newData)
   {    setData(newData);}
   public void setData(int newData)
   {    data = newData;}
   public int getData()
   {    return data;}
}
```



## Example: ListNode (Java)

```
public class ListNode extends Node {
   private ListNode next;
   public ListNode(int newData) {
   //super: access methods of the parent class, including those that
   might be overridden by members of the current class and
   constructors.
        super(newData);
        setNext(null);
   }
   public void setNext(ListNode newNext)
   {    next = newNext;}
   public ListNode getNext()
   {    return next;}
}
```



## Example: TreeNode (Java)

```
pubic class TreeNode extends Node {
   private TreeNode left;
   private TreeNode right;
   public TreeNode(int newData) { // Constructor
        super(newData);
        setLeft(null);
        setRight(null); }
   public void setLeft(TreeNode newLeft)
   {    left = newLeft; }
   public void setRight(TreeNode newRight)
   {    right = newRight; }
   public TreeNode getLeft()
   {    return left; }
   public TreeNode getRight()
   {    return right; }
}
```

## Two Types of Inheritance

- Redefinition (method overriding): refers to the fact that an implementation of a method in a subclass supersedes the implementation of the same method in the base class
  - A derived class is more specific than a base class.

```
public class A
{ public void f1() {System.out.println("A::f1");}}
public class B extends A
{ public void f1() {System.out.println("B::f1");}}
B r = new B();
r.f1();
```



## Two Types of Inheritance

- Redefinition (method overriding): refers to the fact that an implementation of a method in a subclass supersedes the implementation of the same method in the base class
  - A derived class is more specific than a base class.

```
public class A
{ public void f1() {System.out.println("A::f1");}}
public class B extends A
{ public void f1() {System.out.println("B::f1");}}
B r = new B();
r.f1();
Output: B::f1
```



## Class Object (Java)

- In Java, all classes implicitly extend class Object
  class A {…} **means** class A extends Object {…}
- Methods equals and toString.

```
class Object
{...
  //Comparing two objects
  public boolean equals(Object obj) {…}
  //converts a Date object to a string and returns the result
  public String toString() {…}
…}
```



## == vs. equals (Java)

- Strings in Java should always be tested using equals.
- toString function is used by many system utilities to convert an object into printable form.
  - System.out.println implicitly calls the toString method of any object it is asked to print



## == vs. equals (Equal.java)

```
import java.io.*;
public class Equal {
    public static void main(String[] args){
      String s = new String("Hello");
      String t = new String("Hello");
      if(s.equals(t)){
          System.out.println("equal");
      } else {
          System.out.println("not equal");}
      if(s==t){
          System.out.println("s=t");
      } else {
          System.out.println("not(s=t)");}
    }
}
```



## == vs. equals (Equal.java)

```
import java.io.*;
public class Equal {
    public static void main(String[] args){
      String s = new String("Hello");
      String t = new String("Hello");
      if(s.equals(t)){
          System.out.println("equal");
      } else {
          System.out.println("not equal");}
      if(s==t){
          System.out.println("s=t");
      } else {
          System.out.println("not(s=t)");}
    }
}
Output: equal
        not(s=t)
```

## Overriding: equals, toString (Complex2.java)

- Inheritance also allows the behavior of methods to be changed or overridden in subclasses.

```
public class Complex{…
    public boolean equals (Complex c)
     {return re == c.realpart() && im == c.imaginarypart();}
     public String toString() {return re + "+" + im + "i";}
  …
 }
```

```
Complex z = new Complex(1,1);
Complex x = new Complex(1,1);
if (x.equals(z)) System.out.println("ok!");
System.out.println(z);
```



## Overriding: equals, toString (Complex2.java)

- Inheritance also allows the behavior of methods to be changed or overridden in subclasses.

```
public class Complex{…
    public boolean equals (Complex c)
     {return re == c.realpart() && im == c.imaginarypart();}
     public String toString() {return re + "+" + im + "i";}
  …
 }
```

```
Complex z = new Complex(1,1);
Complex x = new Complex(1,1);
if (x.equals(z)) System.out.println("ok!");
System.out.println(z);
```

**Output: ok!**
**1.0 + 1.0i**



## References

- Chapters 5.4, 10.1, 10.2, 10.3, 10.8 in Kenneth Louden's book
- Thinking in java
  http://www.linuxtopia.org/online_books/programming_books/thinking_in_java/index.html
- Inheritance in C++
  http://www.csse.monash.edu.au/~jonmc/CSE2305/Topics/05.09.Inheritance/html/text.html
- Why private inheritance
  http://www.cprogramming.com/tutorial/private.html



## References

- C++ vs Java
  - http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andjava.html
  - http://www.dickbaldwin.com/java/Java008.htm
  - http://www4.ncsu.edu/~kaltofen/courses/Languages/JavaExamples/cpp_vs_java/