CSCI 112
July 1, 2010
Teacher: James Church

Program Design and Data Structures

Program Design – the organization of a program's execution.
Data Structures – the organization and accessing of a record of data

Most of the data types discussed in 111 are those known as the primitives.

Variables – think of data as a block, and information is stored in the block. Variable is a location you can read from and write from
Fields – a variable associated with an object (high level block of memory).
Static Fields – a variable associated with a class
Instance Variables – not associated with a class or object. It is a variable that appears in a block of code. Most of the variables in CS111 are these

Primitive Types – distinct types given to you by java, there are 9 types, eight which store values

- **void** – Has no value
- **boolean** – true/false
- **byte** – -128 to +127 (rarely used, unless application is very memory intensive)
- **character** – 16-bit type, stores characters
- **short** – also a 16-bit type, stores numbers (2^16, so range is -32768 to +32767)
- **int** – 32-bit type, stores numbers (~-2billion to ~+2billion)
- **float** – 32-bit type for storing real numbers (known as low-precision, about 6 digits of accuracy)
- **double** – 64-bit type for storing real numbers (known as high-precision, about 12 digits of accuracy)
- **long** – 64-bit type, stores numbers (~-9quadrillion to ~+9quadrillion)

All variables have **scope. Scope** is defined by the curly brackets {}

Example code to illustrate:

```
class Example {
	public static void main( String[] args) {
		int count;
		{
			int students = 9;
		}
		count = students; //will error: cannot find symbol 'students'
	}
}
```

This error will return because "students" is inside of brackets, so things outside the brackets will not be able to see it. Java will let you make blocks like this for no reason, but there is no practical use for this.

Java Input – as of java 1.5 (java 5), there is a library known as the **Scanner** library.

Scanner – java library used for input. You need to import the scanner with the code:

import java.util.Scanner;

How to use the "Scanner" class:

*Scanner scan = new Scanner(*some source*);*

For the source you can use:

- *System.in* – keyboard input
- *"Hello. My names is..."* - that string
- *new File("*some file name*");* - read from a file, to use this, must import the *java.io.** library

Reading Imput from "Scanner"

- *scan.next()* - reads a string up to first space
- *scan.nextLine()* - reads a string up to new line
- *scan.nextInt()* - reads an integer
- *scan.nextDouble()* - reads a double
- *scan.hasNext()* - will return a true/false if there is another token
- *scan.hasNextLine()* - will return a true/false if there is another line
- *scan.hasNextInt()* - will return a true/false if there is another int
- *scan.hasNextDouble()* - will return a true/false if there is another double

There are more scan methods, but these are the most common.

Example Code (variation of first homework assignment) *Name.java*:

```
import java.util.*;

class Name{
	public static void main( String[] args) {
		Scanner scan = new Scanner(System.in);
		System.out.print("What is your name?");
		String name = scan.nextLine();
		System.out.println("Hello, " + name);
	}
}
```

In the homework assignment, take a string (i.e. "AbCdEfGh") and change the case of the letters.

For this assignment the library *java.lang.Character* will be important

Code:

```
import  java.lang.Character;

class CaseChange{
        public static void main( String[] args ){

                String phrase = "AbCdEfGh";
                char ch = phrase.charAt(0); //first character

                if( ' a ' <= ch && ch <= 'z'){
                        //lower case
                        ch = Character.toUppercase(ch);
                }
                else if( ' A ' <= ch && ch <= 'Z'){
                        //upper case
                        ch = Character.toLowercase(ch);
                }

                System.out.println(ch);
        }
}
```

Loops – two primary types

- Pre-conditional – test, then execute loop
- Post-conditional – execute, then test

Three Looping Structures in Java:

- **for** - pre
- **while** - pre
- **do...while** - post

Code ("for" loop):

```
for (initialization; condition; incremental){
        //loop body
}
```

Any of three can be left blank, but semi-colons will still be required. Initialization will default to do nothing, condition will default to true, incremental will default to do nothing.

```
for(;;); //infinite loop
```

Count to 10:

```
for(int I =1; i<=10; i++){
        System.out.println("#) +i);
}
```

Code ("while" loop):

Count to 10:

```
int i=1;
while(i<=10){
        System.out.println("#"+i);
        i++;
}
```

Code ("do...while" loop):

Count to 10:

```
int i=1;
do{
        System.our.println("#"+i);
        i++;
}while(i<=10);
```

Two Control Works for Loops

- *break;* - immediately end current loop
- *continue;* - start loop over;
  - three caveats to this
    - do...while – starts loop over, but does not test
    - while – retest
    - for – perform incremental and retest

Chapter 7: Arrays

Array: a continuous block of memory representing variables of one data type

- the size of the array is fixed
- the starting index is 0
- to find length of an array: *name.length* (no parentheses on end, as in finding string length)

Code (create an array):

```
int [] jenny = {8, 6, 7, 5, 3, 0, 9];
//indexes:       0, 1, 2, 3, 4, 5, 6

jenny[6] = 8; //write the new value '8' to position 6

int[] numbers = int new[100]; //initializes 100 array elements to the value 0

System.out.println(jenny[0]); //prints 8
System.out.println(jenny[1]); //prints 6
```

```
for(int i=0; i<jenny.length; i++){
        System.out.print("Enter digit for position " + (i+1) + ":");
        jenny[i] = scan.nextInt();
}
```

CSCI 112
Friday, July 2, 2010
Instructor: James Church

Sometimes we need to find different properties of an array.

Basic Algorithm: Finding the max value in a list of integers

```
int[] numbers = new int [100];
.
.       //populate the array with values
.
int maxIndex = 0;
for( int index = 1; index < number.length; index++ ){
        if( number[maxIndex] < numbers[index] ){
                maxIndex = index;
        }
}
int largestValue = numbers[maxIndex];
```

Bounds Checking

```
{ 10, 20, 30, 40, 50 }
  First          Last
 Bound          Bound
```

index inside bounds: legal
index outside bounds: illegal

If you try to access data outside of the bounds, an exception is thrown.

- *ArrayOutOfBoundsException*

Java Methods

Method – set of procedures that can be called as needed.

- so, if you create a method and never call it, that code is never used, you must explicitly call methods to have them execute.

Three Parts of a Method

- **Return Type** – What data type is produced by this method ('void' if no type).
- **Signature** – A combination of names and input parameters.
- **Body** – Code which is executed when the method is called.

Example Method: Area of a Circle

```
static double areaOfCircle( double r ){
//declaring this as a static method means we can call it from other methods
        return 2.14159*r*r;
}
```

Example Method: How to Find the Average of a List of Numbers (Version 1)

```
static double average( int[] numbers ){
        double sum = 0.0;
        for( int i=0; i<numbers.length; i++ ){
                sum = sum + numbers[i];
        }
        return sum/(double)number.length;
}
```

Example Method: How to Find the Average of a List of Numbers (Version 2)

```
static double average( int … numbers ){
// "..." operator allows us to take as many arguments as we want and pack them into an array called
// numbers
        double sum = 0.0;
        for( int I: numbers ){
                sum += I;
        }
        return sum / (double)numbers.length;
}
```

How to Use Method: Average (works for v1 and v2)

```
int[] a = { 10, 20, 30, 40 };
System.out.println( average(a) );
```

How to Use Method: Average (works only for v2)

```
System.out.println( average(10, 20, 30, 40) );
```

Parameter Passing

- **Pass-by-value:** Transfer the value of a variable directly to the method.
- **Pass-by-reference:** Transfer the address of a variable to the method.

For Primitives, Java passes by value.
For Arrays and Objects, Java passes by reference.

The key advantage to pass-by-value is that there is no ambiguity about what is being sent.
Pass-by-reference has the advantage of knowing how many elements there are.

If you are passing by value, and there are 1,000,000 objects being sent, they must all be sent.
If you are passing by reference, and there are 1,000,000 objects being sent, only the one needed is sent.

Java takes this behavior out of your hands, and there is no way to bypass it. Thus, there is no extra syntax to learn.

CSCI 112
Friday, July 2, 2010
Instructor: James Church

Classes

The Class and the Object

Superhero – example of a class
Superman – example of an object

Plato's Theory of Forms

1. Everything is based on a form.
2. That form does not exist in this world.
3. The best we can do is study examples of the form.

Object-Oriented Design Principles

In object-oriented programming we have the opportunity to design our own forms (aka classes).

1. **Modularity –** we can make multiple versions of an object
2. **Abstraction** – there is a layer of abstraction between you and the class. The methods and fields should represent a generic version of the class.
3. **Encapsulation –** possibly the most important part. When trying to design our classes, we want to try and keep how the class works private. The user knows as little about the underlying algorithms and data structures as possible. In other worlds, *keep details private.*

Object-Oriented Goals

1. **Robustness –** software should be able to handle exact inputs, as well as inputs that may not be exactly how we inteneded but are still correct. An example would be how a user might input the date:

   • July 2, 2007
   • Jul 2, 2007
   • 07/02/2007
   • 07/02/07
   • 2 July 2007
   • 07-02-2007
   • 07022007
   • 2007-07-02
   • 20070702

2. **Adaptability** – Ability to quickly change when needed.
3. **Reusability –** the idea of code reuse. Our code should work in as many paridigms as possible.

Designing a Class

- **Class Name** should be a noun
- **Method Names** should be verbs

Public and Private

Private fields – keeps data encapsulated
Publc fields – fields are open. No encapsulation.

Class Example (Bicycle)

Class

```
public class Bicycle {
        public class Bicycle {
                private int gear;
                private int speed;

                // Constructor: first method called in a class
                // Constructor has no return type, is also named same as class
                public Bicycle(){
                        gear = 1;
                        speed = 0;
                }

                // Getter (Accessor) Methods: get data from field
                public int getGear(){
                        return gear;
                }
                public int getSpeed(){
                        return speed;
                }

                // Setter (Mutator) Methods: change data in field
                public void setGear( int gear ){
                        if( gear >= 1 && gear <= 7 ){
                                this.gear = gear;
                        }
                }
                public void setSpeed( int speed ){
                        if( speed >= 0 ){
                                this.speed = speed;
                        }
                }

                // toString method
                public String toString(){
                        return "Gear: " + gear + " Speed: " + speed;
                }
}
```

Test Class

```
import java.util.*;

public class BikeTest{
```

```
        public static void main( String[] args ){
                Bicycle a = new Bicycle();
                Bicycle b = new Bicycle();

                a.setSpeed( 10 );
                a.setGear( 2 );

                b.setSpeed( -10 );  //silent failure, no error message, speed stays where it was
                b.setGear( 3 );

                System.out.println( a );
                System.out.println( b );
        }
}
```

Example Code (Coin.java)

```
public class Coin{
   private final int HEADS = 0;
   private int face;

   //constructor
   public Coin(){
      flip();
   }

   public void flip(){
      face = ( int )( Math.random() * 2 );
   }

   public boolean isHeads(){
      return ( face == HEADS );
   }

   public String toString(){
      return ( face == HEADS )? "Heads" : "Tails"
   }
}
```

CoinTest.java

```
import java.util.*;

public class CoinTest{
   public static void main( String[] args ){
      int headCount = 0;
      int tailCount = 0;
      Coin penny = new Coin();
      final int FLIPS = 100;

      for( int i = 0; i < FLIPS; i++ ){
```

```
            penny.flip();
            System.out.println( i + ": " + penny );

            if( penny.isHeads() ){
               headCount++;
            }
            else{
               tailCount++;
            }
         }
         System.out.println( "Heads: " + headCount );
         System.out.println( "Tails: " + tailCount );
      }
}
```

Static vs. Non-Static

Static is associated with the class.
Non-Static is associated with the object.

Java requires that we always start out in a static context.

You can modify static from non-static, but you can't do things the other way around.

Example Code (MyClass.java)

```
public class MyClass{
   private static int classf;
   private int objectf;

   public MyClass(){
      class f = 10;
      objectf = 10;
   }

   public void reset(){
      classf = 10;
      objectf = 10;
   }

   //class method
   public static void classm(){
      classf = 20;
   }

   //object method
   public void objectm(){
      objectf = 20;
   }

   public void both(){
```

```
        classf = 30;
        objectf = 30;
    }

    public String toString(){
        return "classf: " + classf + " objectf: " + objectf;
    }
}
```

snstest

```
public class snstest{
    public static void main( String[] args ){
        MyClass a = new MyClass();
        MyClass b = new MyClass();

        a.objectm();
        b.classm();

        System.out.println( "a: " + a );
        System.out.println( "b: " + b );

        MyClass.classm();  //also allowed because classm() only affects the static field
    }
}
```

CSCI 112
Wednesday, July 7, 2010
Class Notes

Review of Last Class

Vocabulary

Class – plan/blueprint which defines an object
Object – a collection of variables and associated methods
Method – an action performed on an object
Constructor – a method used to instantiate an object
Parameter – input to a method, sometimes called arguments
Return Type – data type of the method
Private Field – field that cannot be seen outside of the class
Public Field – a field that can be seen by any class
Static Field – associated with the class
Non-Static Field – associated with the object

Subclasses

- Primary example of code reuse.
- **DRY**: don't repeat yourself, keep your code "DRY."
- In a subclass, we are taking a class and making it more specific.

Subclass – derived class, child class
- the new class

Base class – super class, parent class
- old class being extended

IS A – relationship between child and parent classes.

The only way we can derive a class is if the following sentence makes sense:
"Child Class is a Parent Class."
If so, you can successfully make a child class.

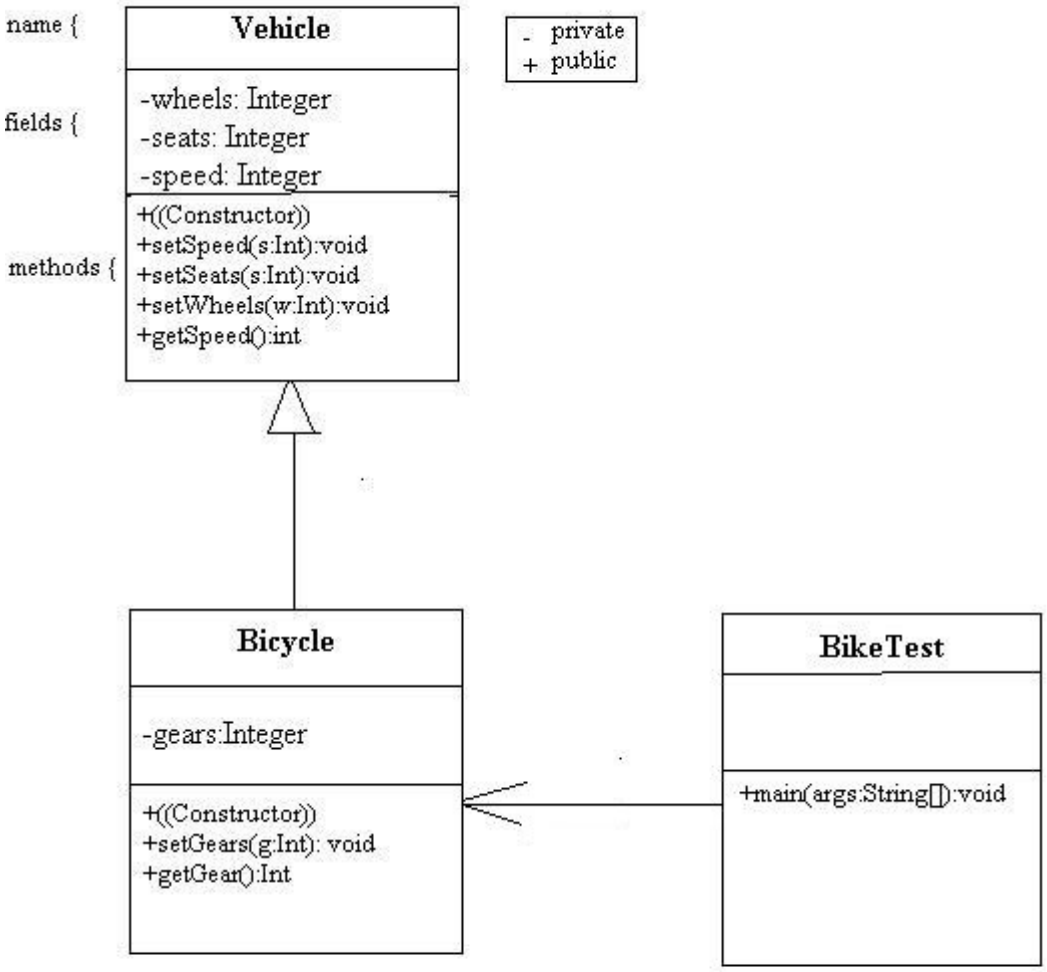
Examples:

Correct: "Horse is a mammal"
Incorrect: "Country is a USA"

Unified Markup Language ("UML")

Two Symbols to Remember in UML diagrams:
- A triangle at the end of a line means **extended class.**
- A line with an arrow means **accesses this class.**

see 100707 Table1.jpg *for example*



100707 Table1.jpg

Public, Private, Protected, Package

**This list is ordered from most restrictive permission to least restrictive permission*

- **Private:** means that only methods within this class can access this method or field.
- **Protected:** means that methods found in this class or classes that extend this class may access this field.
- **Package:** (the default, has no keyword) means that any method in any class found in this directory may access this method or field.
- **Public:** means that any method in any other class can access this method or field.

Example Code (Vehicle.java)

```
public class Vehicle{
	private int wheels;
	private int seats;
	private int speed;

	public Vehicle(){
		this.wheels = 0;
		this.seats = 0;
		this.speed = 0;
	}

	public void setSpeed( int speed ) {
		this.speed = speed;
	}

	public void setSeats( int seats ) {
		this.seats = seats;
	}

	public void setWheels( int Wheels ) {
		this.Wheels = Wheels;
	}

	public int getSpeed(){
		return speed;
	}
}
```

Bicycle.java

```
public class Bicycle extends Vehicle{
```

```
        private int gears;

        public Bicycle( int gears ){
                setWheels( 2 );
                setSeats( 1 );
                this.gears = gears;
        }

        public void setGears( int gears ){
                this.gears = gears;
        }
        public int getGears(){
                return gears;
        }
}
```

BikeTest.java

```
public class Biketest{
        public static void main( String[] args ){
                Bicycle b = new Bicycle( 1);
                b.setGears( 2 );
                b.setSpeed( 10 );

                System.out.println( "Bike Speed: " + b.getSpeed );
        }
}
```

Example of An Extended Class, More Features

Book.java

```
public class Book{
        protected int pages = 1500;
```

```
	//Default constructor, if you don't have one, Java will create one for you
	public Book(){
	}

	//Initialization Contructor.  Initializes the fields.
	public Book( int pages ){
		if( pages >= 1 ){
			this.pages = pages;
		}
	}

	public void setPages( int pages ){
		if( page >= 1 ){
			this.pages = pages;
		}
	}

	public int getPages(){
		return pages;
	}
}
```

Dictionary.java

```
public class Dictionary extends Book{
	private int definitions = 52500;

	//Default Constructor
	public Dictionary(){
		super(); //calls default base constructor
	}

	//Initialization Construtor
	public Dictionary( int pages, int definitions ){
```

```
                super(pages); //call initialization constructor

                if( definitions >= 1 ){
                        this.definitions = definitions;
                }
        }

        public void setDefinitions( int definitions ){
                if( definitions >= 1 ){
                        this.definitions = definitions;
                }
        }

        public int getDefinitions(){
                return definitions;
        }

        public double computeRatio(){
                return (double)definitions/pages;
        }
}
```

Words.java

```
public class Works{
        public static void main( String[] args ){

                //first dictionary
                Dictionary basic = new Dictionary();

                System.out.println( "Basic, pages: " + basic.getPages () );
                System.out.println( "Basic, words: " + basic.getDefinitions() );
                System.out.println( "Basic, WPP: " + basic.computeRatio() );
```

```
        //second dictionary
        Dictionary oxeng = new Dictionary(2000, 80000);


        System.out.println( "Oxford, pages: " + Oxford.getPages () );
        System.out.println( "Oxford, words: " + Oxford.getDefinitions() );
        System.out.println( "Oxford, WPP: " + Oxford.computeRatio() );
    }
}
```

Inheritance in Other Languages

- **Java** is a *single inheritance* language. This means a class can only derive from one other class.
- **C++** is a *multiple inheritance* language. This means a class can derive from multiple classes. This leads to several problems. One main problem in the C++ language is known as the **diamond problem**.

Diamond Problem

Say that class A and B extend from D and class C extends from A and B. If you want to call a method from D from within C, your program doesn't know how to get there (through A or B), so the program crashes.

So Where is Multiple Inheritance Useful?

Say you have a gas engine and an electric engine. A Hybrid engine inherits from both. There is no way to express this in the Java language. So in Java you can not make efficient reuse of code because you can't inherit from both at the same time, requiring you to code all the code from both into Hybrid.

Traits

A third type of inheritance. Ruby is able to make use of this type. Instead of creating a class, you create traits. Then you allow say your hybrid engine to have traits of both gas and electric. A trait can be seen as a sort of crippled class.

This was introduced in Ruby using the relationship "**like a.**"

Java is not able to make use of this either.

CSCI 112
Thursday, July 8, 2010
Class Notes

Review of Last Class

- New class can be derived from current classes using "**extends**."
- Similar terms: base class, **parent class,** super class
- Similar terms: inherited class, **child class**, derived class
- Child Class must hold an "**is-a**" relationship with the parent
- A class can have multiple constructors.
- A constructor with no parameters is the **default constructor**
- To keep a class field private to outside classes, but public to derived classes, make the access modifier "**protected."**
- In a child class, to call the parent's constructor, we use the keyword **super**.
- Java is a single inheritance language, meaning that a class can only be derived from at most one parent class. C++ is a multiple inheritance language, meaning that a class can be derived from multiple classes.

Overriding Methods

When creating a subclass, that class inherits all the methods of the parent class. Sometimes when we create a subclass we want to change the methods that are inherited. We can do this be overriding them. When a method is called, first Java will check all of the methods in the current class and attempt to execute the best match. If none are found, it moves up to the parent class and checks those methods. This process continues until it reaches the topmost class. If none are found in that class, the program errors out.

Example (TestQuote.java):

```
class Quote{
	public void message(){
		System.out.println( "Whatever you are, be a good one. -Abraham Lincoln" );
	}
}
class NewQuote extends Quote{
	public void message(){
		System.out.println( "A nickel ain't worth a dime anymore. -Yogi Berra" );
	}
}
class TestQuote{
	public static void main( String[] args ){
		Quote q = new Quote();
		NewQuote p = new newQuote();
```

```
                q.message();
                p.message();
        }
}
```

Sometimes you don't want a method to be overridden. You show this by using the keyword **final:**

```
public final void message(){}
```

Use this if you're certain that the implementation you create will never need to be overridden.

Abstract Classes

- **Abstract Class:** represents a concept object (i.e. there is no such thing as an animal, there are types of animals).
  - In order to create an abstract class, you prefix it with the keyword abstract.

By itself this isn't very useful, but it has properties that can be passed on to other classes. Thus, it can be used as a building block. Usually it contains a mixture of non-abstract methods and abstract methods.

Abstract classes are useful for creating templates of work.

- **Abstract Methods:** must be overridden by a child class
  - In order to make an abstract method, you prefix the method with the word **abstract** then after the closing parentheses, you end with a semicolon.

Example Code

Work.java

```
abstract public class Work{
        public void result(){
                System.out.println( "Get Paid!" );
        }
        abstract public void action();
}
```

Teaching.java

```
public class Teaching extends Work{
        public void action(){
                System.out.println( "I am teaching." );
        }
}
```

AbstractTest.java

```
class AbstractTest{
	public static void main( String[] args ){
		Teaching t = new Teaching();
		t.action();
		t.result();
	}
}
```

Class Hierarchies

There is no limit to how many times a class can be derived, so a parent class can have as many child classes as you'd like. Those classes can then have their own child classes that inherit from the entire chain above them. However, it doesn't work the other way around, remember, Java is *single-inheritance*.

So, when creating methods, you want to put the methods that have the broadest possible use as high in the chain as you can. This allows for keeping your code "DRY." It also makes for a cleaner class hierarchy.

There is one object that is the master of all other objects, thus highest on any hierarchy. This is known as **Object.** All objects derive from Object. In Object there are six or seven methods, so every object you make in Java has these six or seven methods.

Example Code

Dull.java

```
class Dull extends Object{ //even if you don't write "extends Object," it is done automatically
	public String toString(){
		return "I'm an object.";
	}
}
```

Designing for Inheritance

- Every child class should hold an "is-a" relationship with its parent
- Child classes should always be more specific than their parent class
- Design a class hierarchy around reuse. Put common, broad use functionality as high in the hierarchy as possible.
- Override methods to tailor the functionality of currently written methods
- Avoid using old variable names from class to class
- Each class should be responsible for maintaining its own data.
- It is a good idea to override "toString," even if you don't plan on using it
- Abstract classes are concept classes. They aren't actually created, but they contain functionality

that will be shared among child classes.
- Use access modifiers to prevent breaking encapsulation.

Polymorphism

- **“poly”** → Latin term for 'many'
- **“tics”** → Blood sucking creatures
- **“morph”** → 'form'
- **“polymorphic”** → the ability for a data type to hold many forms

Say we have objects **A, B,** and **C**. If **B** inherits from **A**, and **C** from **B**, **A** can store references from **A**, **B** can store references from **A** and **B**, and **C** can store references from **A**, **B**, and **C**.

- **Polymorphism:** A reference may point to any data type based on its own class of derived from that class.

Example Code

PolyTest.java

```
class Parent{
	public void say1(){
		System.out.println( "Parent" );
	}
	public void say2(){
		System.out.println( "Still parent" );
	}
}
class Child extends Parent{
	public void say1(){
		System.out.println( "Child" );
	}
	public void say3(){
		System.out.println( "Still child" );
	}
}
class PolyTest{
	public static void main( String[] args ){
		Parent p = new Parent(); //see "*Late Binding"
		Child c = new Child();

		p.say1(); // Prints "Parent" to the screen
		p.say2(); // Prints "Still parent" to the screen

		c.say1(); // Prints "Child" to the screen
		c.say2(); // Prints "Still parent" to the screen
		c.say3(); // Prints "Still child" to the screen
```

```
                Parent x = new Child(); //see "*Rules for Polymorphism"

                x.say1(); // Prints "Child" to the screen
                x.say2(); // Prints "Still parent" to the screen
                // x.say3(); // Compile error!
        }
}
```

***Late Binding**

Late Binding is in Java so that you can change the binding of something is Java to something else, necessary for polymorphism.

***Rules For Polymorphism**

- Parent thinks it's a parent class.
- Methods overridden by the child are overridden.
- New methods in the child class are not accessible

CSCI 112
Friday, July 9, 2010

Multidimensional Arrays

Example Code

12x12 Multiplication Table

```
for( int i = 1; i <= 12; i++ ){
	for( int j = 1; j <= 12; j++ ){
		System.out.printf( "%3d ", i * j );
	}
	System.out.println();
}
```

One Dimensional Array Definition

```
datatype[]name = new datatype[size];
```

Two Dimensional Array Definition

```
datatype[][]name = new datatype[first_size][second_size];
```

CSCI 112
Monday, July 12, 2010

How to Read from Files (Review)

Example Code (Incomplete code, won't compile)

```
import java.io*;
import java.util.*;

Scanner scan = null;
try{
        scan = new Scanner( new File( filename ) );
}
catch( Exception e ){
        System.out.println( "File not found: " + filename );
        System.exit(1);
}

int[][] a = new int[9][9];

for( int i = 0; i < 9; i++ ){
        for( int j = 0; j < 9; j++ ){
                a[i][j] = scan.nextInt();
        }
}
```

Polymorphism

Three Properties of Polymorphism

```
A a = new B(); //B extends A
```

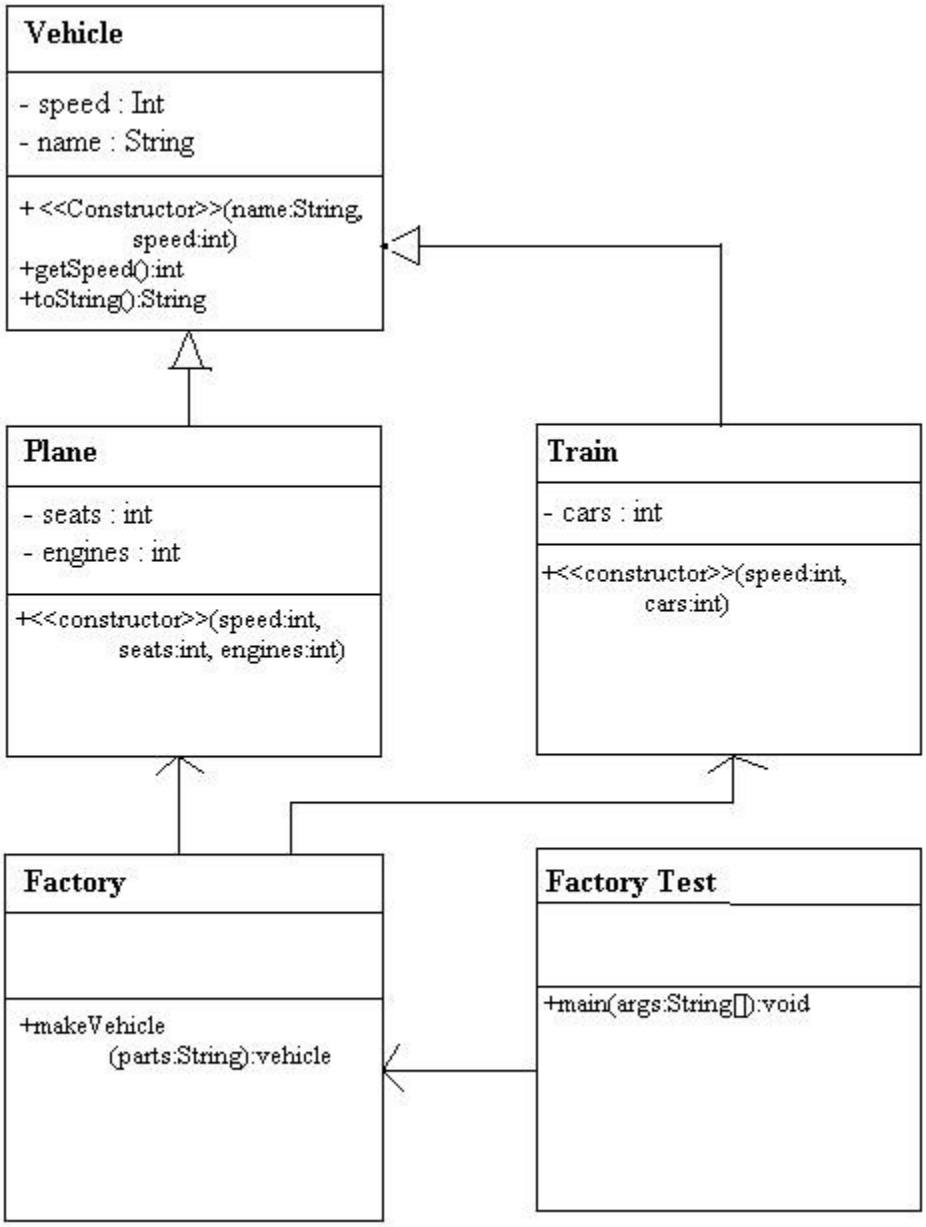
1. "a" may execute all methods in A.
2. "a" will execute overridden methods in B.
3. Methods unique to B are not accessible.

Late Binding – An object is associated with a reference at execution time, not compile time.

The Factory Pattern

- A design pattern developed in 1995 by a group known as the "Gang of Four."
  - The wrote a book called *Design Patterns*.

100712 Table1.jpg – UML Diagram for Example Code



Example Code (Factory Pattern)

parts.txt

```
4
Plane 300 50 2
Train 50 20
Plane 100 2 1
Train 30 10
```

Factory.java

```
import java.util.*;

public class Factory{
	public static Vehicle makeVehicle( String parts ){
		Scanner type = scan.next();

		if( type.equals( "Train" ) ){
			int speed = scan.nextInt();
			int cars = scan.nextInt();

			return new Train( speed, cars );
		}
		if( type.equals( "Plane" ) ){
			int speed = scan.nextInt();
			int seats = scan.nextInt();
			int engines = scan.nextInt();

			return new Plane( speed, seats, engines );
		}
		System.out.println( "Unknown: " + type );
		System.exit(1);
		return new Train( 0, 0 ); //never called
	}
}
```

FactoryTest.java

```
import java.util.*;
import java.io.*;

public class FactoryTest{
	public static void main( String[] args ) throws Exception{
		Scanner scan = new Scanner( System.in );
		System.out.println( "Enter a filename: " );
		String filename = scan.next();

		Scanner fh = new Scanner( new File( filename ) );
		int vehicle_count = fh.nextInt();
		fh.nextLine();

		Vehicle[] vehicles = new Vehicle[vehicle_count];

		for( int i = 0; i < vehicle_count; i++ ){
			String parts = fh.nextLn();

			vehicles[i] = Factory.makeVehicle(parts);
		}
```

```
		for( int i = 0; i < vehicle_count; i++ ){
			System.out.print( vehicles[i] + " " + vehicles[i].getSpeed() );
		}
	}
}
```

Vehicle.java

```
//make code from UML diagram if needed
```

Train.java

```
//make code from UML diagram if needed
```

Plane.java

```
//make code from UML diagram if needed
```

Interfaces

Interface – a collection of constants and abstract methods.

Example Code (Simple Cryptography Program)
Encryptable.java

```
public interface Encryptable{
	public void encrypt();
	public void decrypt();
}
```

Rot13.java

```
public class Rot13 implements Encryptable{
	private String message;
	private boolean encrypted;

	public Rot13( String message){
		this.message = message;

		encrypted = false;
	}
	public String toString(){
		return message;
	}
	private void rotate(){
		String masked = "";

		for( int i = 0; i < message.length(); i++ ){
```

```
                        char ch = message.charAt(i);

                        if( 'a' <= ch && ch <= 'z' ){
                                ch = (char)( 'a' + (( ch - 'a' ) + 13 )%26 );
                        }
                        if( 'A' <= ch && ch <= 'Z' ){
                                ch = (char)( 'A' + (( ch - 'A' ) + 13 )%26 );
                        }
                        masked += ch;
                }
                message = masked;
        }
        public void encrypt(){
                if( encrypted == false ){
                        rotate();
                        encrypted = true;
                }
        }
        public void decrypt(){
                if( encrypted == true ){
                        rotate();
                        encrypted = false;
                }
        }
}
```

RotTest.java

```
public class RotTest{
        public static void main( String[] args ){
                String message = "Attack at Dawn";
                Rot13 r = new Rot13( message );

                System.out.println( r );

                r.encrypt();
                System.out.println( r );
        }
}
```

Interfaces were designed to allow for multiple interfaces to be implemented within one class, because it wouldn't be the same as inheritance and two different interfaces shouldn't contradict each other. One way they can conflict though is if the interfaces contain constants.

Example Code

InterfaceTest.java

```
interface Ia{ final public int X = 2; }
interface Ib{ final public int X = 3; }

class A implements Ia, Ib{
	static void getX(){
		System.out.println(X);
	}
}
class InterfaceTest{
	public static void main( String[] args ){
		A.getX();
	}
}
```

This will not compile, will error: “Cannot compile program, X is ambiguous.” If you comment out one of the interfaces though, it will run just fine.

CSCI 112
Tuesday, July 13, 2010

The Game of Life

Rules:

1. If someone has only one friend, they die
2. If someone has more than three friends, they die
3. If an empty cell has three friends, a child is born

We didn't really have class today, just checked out Game of Life algorithms and Tower of Hanoi algorithms then took a test.

CSCI 112
Wednesday, July 14, 2010

Exception Handling

- handling the program when things go wrong.

Sometimes errors are avoidable, you can write things so that things just won't happen. Other times things are unavoidable, and the user is able to type things in that go wrong.

What can go wrong?

- **division by zero:** any number divided by zero will abort your program.

```
4 / 0
```

- **Array index out of bounds:** try to access an array element out of the array, your program will abort

```
a[-1]
```

- **Accessing files which cannot be found:** if a file can't be found, it will halt your program.

```
new File( “does_not_exist.txt” );
```

- **Following null references:** References point to objects, and when they don't point to anything, they point to “null” this is known as a “null pointer exception.

```
Scanner scan = null;
scan.nextInt();
```

- **Reading an integer using characters:**

```
Integer.parseInt( “Apple” );
```

Three Ways to Handle Exceptions

1. Do nothing, maybe the problem will resolve itself. “When a Java program encounters an exception that isn't handled, it will abort the program and print a “Stack Trace” to the screen. This is handy if you are debugging, because it will report all of the method calls from the start of the program to the current method and which line number called which method.”
2. Pass the problem to another method.
3. Handle the problem right now.
    - Fixing the issue (assuming you know how)
    - Abort the program and explain why the issue halts the program.

Pass the Problem to Another Method

Passing a problem to the caller method is the only way to pass a problem to another method. This is known as **throwing** the exception. You have to specify in the method name that there could be a problem and this method will throw exceptions.

```
void myMethod() throws Exception{
	//statements
	if(/*test for bad conditions*/){
		throw new Exception( "Bad Condition Happened." );
	}
}
```

Handle the Problem Right Now

```
void myMethod(){
	try{
		//statements
		if( /*test for bad condition*/ ){
			throw new Exception( "Bad condition happened." );
		}
	}
	catch( Exception e ){
		System.out.println( "Caught the exception" + e );
		System.exit(1);
	}
}
```

How to catch a problem in Scanner's "nextInt()" method:

```
Scanner scan = new Scanner( System.in );

int n = 0;

while(true) {
	try{
		n = scan.nextInt();
		break;
	}
	catch( InputMismatchException e ){
		System.out.println( "Didn't type a number." );
	}
}
System.out.println( "You typed number " + n );
```

To better illustrate the exception handling traits in Java, this good example is found in the book:

```
// From "Java Program" by Malik, 1-4188-3540-4
// Chapter 12, Page 773
//

import java.io.*;

public class PrintStackTraceExample1 {
   public static void main(String[] args) {
      try {
         methodA();
      }
      catch (Exception e) {
         System.out.println(e.toString() + " caught in main");
         e.printStackTrace();
      }
   }

   public static void methodA() throws Exception { methodB(); }

   public static void methodB() throws Exception { methodC(); }

   public static void methodC() throws Exception {
      throw new Exception("Exception generaged in method C");
   }
}
```

Checked vs. Unchecked Exceptions

Checked Exception – requires that it must be caught of thrown.

- include exceptions related to file I/O

Unchecked Exception – may be ignored

- Classes that extend from RuntimeException are unchecked.

- All other exceptions are checked.

How to Check if an Exception is Checked or Unchecked

Making your own exceptions simply requires making a new class and extending Exception. Make the class name the name of the exception and make the constructor accept a String parameter and then call "super" on that parameter.

```
class MyOwnException extends Exception {
    MyOwnException(String s) {
        super(s);
    }
}
```

Using your exception class like you would any other exception error.

```
void myMethodWhichDoesStuff() throws MyOwnException {
    // Statements
    if ( /* bad condition */ )
        throw new MyOwnError("My Own Error Exception");
}
```

IO Exceptions

The System object.

Three static fields in System:

- they are public
- they are final

1. in → InputStream, reads from keyboard.
2. out → PrintStream, prints to the screen.
3. err → PrintStream, prints to the screen (err is short for error).

```
class PrintTest{
        public void static main( String[] args ){
                System.out.println( "Hello." );
                System.err.println( "There is a problem." );
        }
}
```

If you were to execute this you would see printed to the screen:

```
Hello.
There was a problem.
```

It is possible to run it in the command line without printing the error messages to the screen.

Writing To a File

- causes a checked exception

```
import java.io.*;

class FilewriterTest{
	public static void main( String [] args ) throws IOException{
		String filename = "data.txt";
		FileWriter fw = new FileWriter( finename );
		BufferedWriter bw = new BufferedWriter( fw );
		PrintWriter outfile = new PrinterWriter( bw );

		outfile.println( "Hello, textfile." );

		outfile.close(); //essential to make sure file is properly closed
	}
}
```

CSCI 112
Thursday, July 15, 2010

Analysis of Algorithms

There are two metrics for measuring the efficiency of an algorithm.

- **Time**
- **Space**

The way to record time in java is you record the beginning system time as a *long* variable, then do the same with the system time at the end. The difference of the two is the length of time it took to run.

Example Code: Timing An Algorithm

```
long start = System.currentTimeMillis();
//some operations
System.out.println( "Runtime: " + ( System.currentTimeMillis() - start ) + " milliseconds");
```

Because all computers have different speeds, this isn't an accurate benchmark between systems.

Big O Notation

Study of the number of basic operations performed by an algorithm

Primitive Operations:

- basic assignment: x=1;
- calling methods
- arithmetic
- comparison of two numbers
- indexing into an array
- dereferencing objects
- return from a method

Seven Basic Growth Functions (from fastest to slowest):

- **O(1) – Constant Time**
- **$O(\log_2 N)$ – Logarithmic Time**
- **O(N) – Linear Time**
- **$O(N*\log_2 N)$ – Log-Linear Time**
- **$O(N^2)$ – Square Time**
- **$O(N^3)$ – Cubed Time**
- **$O(2^n)$ – Exponential Time**

Constant Time Algorithms - O(1)

```
int sum( int x ) {
        return ( x * ( x + 1 ) ) / 2;  // 3
}
```

This total basic operations for this code is: 3

Our Big O Notation shows this to be in **O(1)** time. This algorithm has **constant** time.

Logarithmic Algorithms - O($\log_2$N)

This is the second fastest type of algorithm we can have. If you have a range of 100 elements the most choices you have before finding what you want (assuming they are in order) is $\log_2 100$ which is ~8 choices.

Linear Algorithms - O(N)

```
int sum( int n ) {
        int sum = 0;  // 1
        for( int i = 1; i <= n; i++ ){ // 1, n, n, 2n
                sum+=i;
        }
}
```

The total basic operations of this code is: 4n+2

So our Big O Notation shows this to be in **O(N)** time. This algorithm has l**inear** time.

Log-Linear Algorithms - O(N*$\log_2$N)

You don't see these very often, but two very famous algorithms that use this are:

- quicksort
- mergesort

Square Time Algorithms - O(N^2)

For every element in a list, touch every other element. This means if you have 100 elements, you do 10000 operations. Most sorting algorithms fit into this class.

- insertion sort
- selection sort
- bubble sort

Cubed Time – O(N^3)

- Matrix Multiplication

Exponential Time – $O(2^n)$

Won't be seen very often, not commercialized, as they can only be used for very specific problems that can't be solved any other way.

The Clock Game (The Number Guessing Game)

The Clock Game is logarithmic time

On the game show, “The Price is Right,” Bob Barker played a game called, “The Clock Game.” In this game, Bob told the contestant that the value of “the item up for bid” was between a high and low value. The user then had to guess the value of the item within 30 seconds. After each guess, Bob would say “higher” if the guess was too low or “lower” if the guess was too high. It is the only game on “The Price is Right” where it is possible to win every time without knowing anything about the product as long as the contestant follows a logarithmic time algorithm.

Psuedocode

```
low <- Bob's Low Bound
high <- Bob's High Bound
secret <- Bob's Secret Value
while (clock is ticking)
   guess <- (high + low) / 2
   if (secret < guess)
      Bob Says "LOWER!"
      high <- guess + 1
   end if
   if (secret > guess)
      Bob Says "HIGHER!"
      low <- guess + 1
   end if
   if (secret > guess)
      Bob Says "RIGHT!"
      break
   end if
end while
```

Binary Search

This code is a binary search algorithm that essentially plays the clock game.

```
int BinarySearch( int[] nums, int value ){
        int low = 0;
        int high = nums.length - 1;

        while( low <= high){
                int guess = ( high - low ) / 2 + low;
```

```
                if( value < nums[guess] ){
                        high = guess - 1;
                }
                else if( value > nums[guess] ){
                        low = guess + 1;
                }
                if( value == nums[guess] ){
                        return guess;
                }
        }
        return -1;
}
```

Basic Sorting

Bubble Sort

- Group every pair of side by side numbers into overlapping bubbles.
- For each bubble, if it is out of order, swap the values.
- Repeat until all bubbles no longer require swapping.

```
void BubbleSort(int nums){
        for( int i = 0; i < nums.length - 1; i++ ){
                boolean swapped = false;

                for( int j = 0; j < nums.length - 1; j++ ){
                        if( nums[j] > nums[j+1] ){
                                int temp = nums[j];
                                nums[j] = nums[j+1];
                                nums[j+1] = temp;
                                swapped = true;
                        }
                }
                if( swapped == false ){
                        break;
                }
        }
}
```

CSCI 112
Friday, July 16, 2010

Sorting Objects

Let's say we create some Dogs:

```
class Dog{
	private String name;
	private int age;

	public Dog( String name, int age ){
		this.name = name;
		this.age = age;
	}
}
```

We create two dogs (incomplete code):

```
Dog a = new Dog( "Fido", 4 );
Dog b = new Dog( "Odie", 2 );
```

We can't sort the dogs by age the way it is now, because age is private, so we'd have to have getter methods, or we can use the Java libraries. So, let's rewrite *Dog*:

```
class Dog implements Comparable{
	private String name;
	private int age;

	public Dog( String name, int age ){
		this.name = name;
		this.age = age;
	}
	public int compareTo( Object other ){
		Dog d = (Dog)other;
		return this.age - d.age;
	}
}
```

Three Comparable States

With the *compareTo* object, we need to return one of three things:

- if the objects are equal:
  - *return 0;*
- if this is greater than other:
  - return a positive number
- if this is less than other:

- return a negative number

Now that we have this in our class, we can do this:

```
Dog a = new Dog( "Fido", 4 );
Dog b = new Dog( "Odie", 2 );
System.out.println( a.compareTo(b) );
//this will return 2, as it is 4 - 2
```

Or, in a larger example:

```
import java.util.Arrays;

Dog[] myDogs = new Dogs[5];

myDogs[0] = new Dog( "Fido", 4 );
myDogs[1] = new Dog( "Odie", 2 );
myDogs[2] = new Dog( "Lassie", 3 );
myDogs[3] = new Dog( "Spot", 1 );
myDogs[4] = new Dog( "Buddy", 5 );

Arrays.sort(myDogs);
//uses quick sort, able to do because our class Dog implements Comparable
```

Bubble Sort

Algorithm

```
void bubble_sort( int[] a ){
        while( true ){
                boolean swapped = fale;
                for( int i = 0; i < a.length - 1; i++ ){
                        if( a[i] > a[i+1] ){
                                int temp = a[ii];
                                a[i] = a[i+1];
                                a[i+1] = temp;
                                swapped = true;
                        }
                }
                if( swapped == false )
                        break;
        }
}
```

Gap Sort

- Based on the bubble sort

Algorithm:

1. Start with a large gap

```
gap = array.length -1;
```

2. Compare all elements “gap” units apart
3. Reduce 'gap' by 1
4. go back to beginning
5. gap of 1 means one pass of the bubble sort

Code:

```
void swap( int[] a, int i, int j){
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}
void gap_sort( int[] a ){
        for( int gap = a.length - 1; gap >= 1; gap-- ){
                for int i = 0; i + gap < a.length; i++ ){
                        if( a[i] > a[gap + i] ){
                                swap( a, i, gap+i );
                        }
                }
        }
}
```

Drunk Sort

- The most useless of sorts, but it's fun to know

```
void swap( int[] a, int i, int j){
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}
void drunk_sort( int[] a ){
        Random rng = new Random();
        boolean sorted = true;

        while( true ){
                for( int i = 0; i < a.length; i++ ){
```

```
                        swap( a, i, rng.nextInt( a.length ) );
                }
                sorted = true;
                for( int i = 0; i < a.length - 1; i++ ){
                        if( a[i] > a[i+1] ){
                                sorted = false;
                        }
                }
                if( sorted )
                        break;
        }
}
```

CSCI 112
Monday, July 19, 2010

Counting Sort

- Very fast sort
  - works in **O(N+S)** time
  - **N**: number of elements to sort
  - **S**: range of elements from smallest to largest
- Works best when the range of values is small.

Code:

```
void countingSort( int[] a ){
        int min = a[0];
        int max = a[0];

        for( int i = 1; i < a.length; i++ ){
                if( a[i] > max )
                        max = a[i];
                if( a[i] < min )
                        min = a[i];
        }
        int range = max - min + 1;

        int[] count = new int[range];

        for( int i = 0; i < a.length; i++ ){
                count[a[i] - min]++;
        }

        int z = 0;

        for( int i = min; i <= max; i++ ){
                for( int j = 0; j <= count[i-min]; j++ ){
                        a[z++]=i;
                }
        }
}
```

Selection Sort

- done in $\mathbf{O(N^2)}$ time

1. Find the location of the smallest value in the unsorted portion.
2. Swap the location of the minimum with the first element in the unused portion.

Pseudo-Code

```
function selection_sort( int[] a )
        for i = 0 to length(a)
                min_position = i
                //find the min

                for j = i+1 to length(a)
                        if a[j] < a[min_position]
                                min_position = j
                end for
                //swap values
                temporary = a[i]
                a[min_position] = temporary
        end for
end function
```

Insertion Sort

- done in **$O(n^2)$** time

Three Parts of Insertion Sort

1. Sorted Portion
   - At the beginning, this is the first element
2. Next Number to Sort
   - At the beginning, this is the second element
3. Unsorted Portion
   - At the beginning, this contains everything after the second element
   -

Basic Steps for the Insertion Sort

- If the value to be inserted is larger than the largest portion of the sorted list, do nothing.
- If not, do the following:
  1. Store the value in a temporary and
  2. Starting with the tail, move values to the end one value at a time until the gap is created which will house our number.
  3. Take the value out of the temporary and add it to the gap.

Pseudo-Code

```
function insertion_sort( int[] a )
        for i = 1 to length(a) - 1
                temporary = a[i]
                low_position = i - 1
```

```
            while low_position >= 0 and temporary < a[low_position]
                a[low_position+1] = a[low_position]
                low_position = low_position - 1
            end while
            a[low_position+1] = temporary
        end for
end function
```

Shell Sort

- Improvement of the Insertion Sort
- runs in **O()** time

Steps for Shell Sort

Start with this list:

```
5 3 8 4 5 2
```

1. Divide the list in two

```
5 3 8
4 5 2
```

2. Perform insertion sort on each column

```
4 3 2
5 5 2
```

3. Divide list into three
4. Perform insertion sort on each column
5. Divide the list into four
6. Perform insertion sort on each column

Code

```
void gapInsertionSort(int[] a; int start, int gap){
        for(int i = start+gap; i < a.length; i++){
            int temp = a[i];
            int loaw_position = i - gap;
            while( low_position >= start && temp < a[low_position] ){
                a[low_position+gap] = a[low_position];
                low_position -= gap;
            }
            a[low_position+gap] = temp;
        }
}
void shell_sort( int[] a ){
        for(int gap = a.length/2; gap>=1; gap--){
            for( int i = 0; i <= gap; i++ ){
                gapInsertionSort(a, i, gap);
```

```
                }
            }
}
```

CSCI 112
Tuesday, July 20, 2010

Collections and Stacks

Collections

Collection: A data structure used to hold an unspecified number of data elements.
- Unlike an array, the size does not need to be declared at the declaration.

Two Types of Collections:

Linear: Stores data sequentially.
Non-Linear: Stores data as a network, grid, hierarchy, or has no order at all.

Basic Operations of a Collection

- Add elements
- Remove elements

Collections have a starting size of zero.

Goals of a Collection

Abstraction: Details are hidden.
Encapsulation: Data access is restricted.

Good Questions to Ask When Creating a Collection

- How will the collection operate?
- What will the interface look like?
- What problems will the collection solve?
- How will we implement the interface?
- What are the benefits and penalties to each implementation?

Generics

Generic: The ability to create a placeholder type when the type the user will want is not known.

To do this, we use the generic **T** type.

Java's generics require that T be a class.

Code

```
class Myclass <T> {
```

```
        T a;

        void set(T a){
                this.a = a;
        }
        T get(){
                return a;
        }
}
```

Code

```
public void static main{ String[] args ){
        MyClass<String> x;
        x.set( "Hello" );

        MyClass<Integer> y;
        y.set( new Integer(21) );
}
```

Stack

Stack: a collection

Stacks are **LIFO** data structures

- **LIFO:** Last-In-First-Out, the last data value put on the stack is the first data value off the stack

Applications of the Stack

- Operating Systems: concept known as the Stack Pointer
- The undo button on your text editor.
- The back button on your web browser

Six Stack Operations

- **boolean:** *isEmpty()* : returns true if empty, returns false otherwise
- **boolean:** *isFull()* : returns true if full, returns false otherwise
- **int:** *size()* : returns current number of elements on the stack
- **void:** *push(T data)* : adds a value to the top of the stack
- **T:** *pop()* : removes data from the top of the stack
- **T:** *peek()* : returns top element without removing it from the stack

Stack Implementation

Stack.java

```
class Stack<T>{
	final private int MAX_SIZE = 100;

	T[] elements;

	int top;

	public Stack{
		elements = (T[])new Object[MAX_SIZE];
		top = 0;
	}
	public boolean isEmpty(){
		return ( top == 0 );
	}
	public boolean isFull(){
		return ( top == MAX_SIZE )
	}
	public int size(){
		return top;
	}
	public T peek(){
		if( isEmpty() ){
			System.out.println( "Error: stack empty" );
			System.exit(1);
		}
		return elements[top-1];
	}
	public void push( T data ){
		if( isFull() ){
			System.out.println( "Error: stack is Full." );
			System.exit(1);
		}
		element[top] = data;
		top++;
	}
	public T pop(){
		if( isEmpty() ){
			System.out.println( "Error: stack empty" );
			System.exit(1);
		}
		top--;
		return elements[top];
	}
}
```

StackTest.java

```
public class StackTest{
	public static void main( String[] args ){
		Stack<String> s = new Stack<String>();

		s.push( "Hello." );
		s.push( "Hi." );
		s.push( "Howdy." );
		s.pop()
		System.out.println(s.size());
		System.out.println(s.pop());
	}
}
```

CSCI 112
Wednesday, July 21, 2010

Nodes

```
class MyClass{
	public String s = "Hi";
}

class Node<T> {
	public T data;
	public Node next;
}
class Main{
	public static void main( String[] args ){
		Node<String> head = new Node<String>();

		head.data = "A";

		head.next = new Node<String>();
		head.next.data = "B";

		head.next.next = new Node<String>();
		head.next.next.data = "C";

		head.next.next.next = null;
}
```

Traversing a List of Nodes

```
Node<String> ptr = head;

while( ptr != null ){
	System.out.println( ptr.data );
	ptr = ptr.next;
}
```

CSCI 112
Thursday, July 22, 2010

Nodes

Node – a node has two parts

1. Data portion
2. Link portion

A list of nodes is a "**Linked List**."

StackTest.java

```
class StackTest{
	public static void main( String[] args ){
		Stack<Integer> s = new Stack<Integer>();

		s.push(10);
		s.push(20);
		s.push(30);
		s.push(40);
		s.push(50);

		while( s.isEmpty == false ){
			System.out.println( s.pop() );
		}
	}
}
```

Stack.java

```
class Stack<T>{
	private class Node{
		public T data;
		public Node next;
	}

	private Node head;
	private int size;

	public Stack(){
		size = 0;
		head = null;
	}
	public boolean isEmpty(){ return size == 0; }
	public int size(){ return size; }
	public void push(T data){
```

```
                Node myNode = new Node();
                myNode.data = data;
                myNode.next = head;
                head = myNode;
                size++;
        }
        public T pop(){
                if( isEmpty() ){
                        System.out.println( "Error: Just like Pringles, you can only pop until they're all
gone." );
                        System.exit(1);
                }
                size--;
                T myData = head.data;
                head = head.next;
                return myData;
        }
        public T peek(){
                if( isEmpty() ){
                        System.out.println( "Error: You're out of stuff to peek at." );
                        System.exit(1);
                }
                return head.data;
        }
}
```

Queues

While Stacks follow the **LIFO** model, Queues follow the **FIFO (First-In-First-Out)** model.

Two Basic Operations of a Queue

enqueue → add to the back of the list
dequeue → remove from the front of the list

Typically you will also have another method known as **front**, which looks at the first element.

Queue Implementation Algorithm (Pseudo-code):

```
class Queue<T>
   int MAX_SIZE = 100
   int front
   int back
   T[] elements

   Queue:
      front = 0
```

```
    back  = 0
    elements = Array[MAX_SIZE]

isEmpty:
    return front == back

isFull:
    return front == (back + 1)%MAX_SIZE

size:
    return (back - front + MAX_SIZE) % MAX_SIZE

front:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    return elements[front]

enqueue(data):
    if isFull:
        print "Error: Queue is full."
        exit
    elements[back] = data
    back = (back + 1) % MAX_SIZE

dequeue:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    temp = elements[front]
    front = (front+1) % MAX_SIZE
    return temp
```

Queue Implementation: Array

```
class Queue<T>{
        private final int MAX_SIZE = 100;
        private int front = 0;
        private int back = 0;
        private T[] element;

        public Queue{
                front = 0;
                back = 0;
                elements = (T[] new Object[MAX_SIZE];
        }
        public boolean isEmpty(){
                return front == back;
        }
```

```
	public boolean isFull(){
		return front == (back+1)%MAX_SIZE;
	}
	public int size(){
		return (back-front+MAX_SIZE)%MAX_SIZE);
	}
	public T front(){
		if(isEmpty()){
			System.out.println( "Error: empty" );
			System.exit(1);
		}
		return elements[front];
	}
	public void enqueue(T data){
		if(isFull()){
			System.out.println( "Error: full" );
			System.exit(1);
		}
		element[back] = data;
		back = (back+1)%MAX_SIZE;
	}
	public T dequeue(){
		if(isEmpty()){
			System.out.println( "Error: empty" );
			System.exit(1);
		}
		T temp = elements[front];
		front = (front+1)%MAX_SIZE;
		return temp;
	}
}
```

Priority Queue

Priority Queue – a cross between a queue and an insertion sort. When enqueue'ing data, you perform an insertion sort to decide where to put it.

Dijkstra's Algorithm – shortest distance solver, an example of a priority queue.

Node Insertion

Two Problems

- insert at beginning
- insert beyond the beginning

Two Pieces of Information to Insert

- The value to be inserted
- the position

Insert at Beginning

1. Make a new node to hold data.
2. Set the new node's next field to head
3. Set the head to the new note.

Insert Beyond the Beginning

1. Make a new node to hold the data.
2. Traverse the list until you reach the node prior to the insertion point.
   - thisNode → node prior to insertion point
   - myNode → node to be inserted
3. thisNode's next field should be copied to myNode's next field
4. thisNode's next field should now point to myNode

Deleting a Node/Node Removal

Two Problems

- delete from beginning
- delete beyond beginning

One Piece of Information

- position to delete

Delete from Beginning

1. Store head.data field in temporary
2. head should point to head's next field

Delete Beyond the Beginning

1. Traverse the list until you reach the node prior to the deletion point.
   - thisNode → node prior to deletion point
2. Save thisNode.next.data in a temporary.
3. ThisNote.next = thisNode.next.next

CSCI 112
Friday, July 23, 2010

Homework

LinkedList.java

```
class LinkedList{
	private class Node{
		public int data;
		public Node next;

		public Node(){
			data = 0;
			next = null;
		}
		public Node(int data){
			this.data = data;
			next = null;
		}
	}
	private Node head;
	private int size;

	public LinkedList(){
		head = null;
		size = 0;
	}
	public boolean isEmpty(){
		return size == 0;
	}
	public int size(){
		return size;
	}
	public int get(int pos){
		if( pos < 0 || pos >= size ){
			System.out.println( "Error: Out of Bounds: " + pos );
			System.exit(1);
		}
		Node thisNode = head;
		for( int i = 0; i < pos; i++ ){
			thisNode = thisNode.next;
		}
		return thisNode.data;
	}
	public void set(int pos, int data){
```

```
        if( pos < 0 || pos >= size ){
                System.out.println( "Error: Out of Bounds: " + pos );
                System.exit(1);
        }
        Node thisNode = head;
        for( int i = 0; i < pos; i++ ){
                thisNode = thisNode.next;
        }
        thisNode.data = data;
}
public int remove(int pos){
        if( pos < 0 || pos >= size ){
                System.out.println( "Error: Out of Bounds: " + pos );
                System.exit(1);
        }
        int value; //stores data being removed
        if(pos == 0){
                value = head.data;
                head = head.next;
        }
        else{
                Node thisNode = head;
                for( int i = 0; i < pos-1; i++ ){
                        thisNode = thisNode.next;
                }
                value = thisNode.next.data;
                thisNode.next = thisNode.next.next;
        }
        size--;
        return vlue;
}
public void insert(int pos, int value){
        if( pos < 0 || pos > size ){
                System.out.println( "Error: out of bounds: " + pos );
                System.exit(1);
        }
        Node myNode = new Node(value);
        if( pos == 0 ){
                myNode.next = head;
                head = myNode;
        }
        else{
                Node thisNode = head;
                for( int i = 0; i < pos-1; i++ ){
                        thisNode = thisNode.next;
                }
                myNode.next = thisNode.next;
                thisNode.next = myNode;
        }
```

```
                size++;
        }
}
```

Applications of Queues

Cryptography

Logic Tables

“And” Logic:

A	B	And(A, B)
0	0	0
0	1	0
1	0	0
1	1	1

“Or” Logic:

A	B	Or(A, B)
0	0	0
0	1	1
1	0	1
1	1	1

“Not” Logic

A	Not(A)
0	1
1	0

“XOR” Logic:

A	B	XOR(A, B)
0	0	0
0	1	1
1	0	1
1	1	0

Code:

```
int a = 2;
int b = 5;
int c = a ^ b; //a XOR b
System.out.println(c); //6
```

How Does this Relate to Queues?

Code:

```
Queue<Character> q = new Queue<Character>();

String message = "Attack at dawn.";
String key = "key";

for( int i = 0; i < key.length(); i++ ){
        q.enqueue(ket.charAt(i));
}
for( int i = 0; i < message.length(); i++ ){
        int a = (int)q.dequeue();
        int b = (int)message.charAt(i);
        System.out.println( (char)(a^b) );
        q.enqueue((char)a);
}
```

CSCI 112
Monday, July 26, 2010

Linked Lists

Using a Linked List to Create a Stack and Queue

```
class Stack<T> extends LinkedList{
        public Stack{ super(); }
        public void push(T data){ super.insert(0, data); }
        public T pop(){ return super.remove(0); }
        public T peek{ return super.get(0); }
}
```

A Stack is not a Linked List:

- A Linked List provides additional functionality that a Stack would not be able to do

```
class Stack<T> {
   private LinkedList<T> elements;

   public Stack() { elements = new LinkedList<T>(); }
   public boolean isEmpty() { return elements.size() == 0; }
   public boolean isFull() { return false; }
   public int size() { return elements.size(); }
   public void push(T data) { elements.insert(0, data); }
   public T pop() { return elements.remove(0); }
   public T peak() { return elements.get(0); }
}
```

Doubly-Linked Lists

Let's review the basic interpretation of a Node:

- A data field.
- A node reference pointing to the next data node (or this is null if there is no next data field)
- A double linked list requires an additional field to the idea of a Node: one node reference pointing to the element that should go before this node, or null if there is no previous node.

The major downside to a doubly linked list is that insertions and deletions (which are already complicated in the singly linked list implementation) are now even more complicated.

The upside to a doubly linked list is speed. In addition to having a Node link always point to the front

of the list (or “head”), there is also a Node link pointing to the back of the list (or “tail”). If you have a N elements (where N is a very large number) in a singly linked list, it takes N operations to get the value of the last node. In a doubly linked list, the operation time is O(1) to grab the last element.

For most algorithms requiring a set of data that grows, the operations of insertion and removal happen on the end extremities. By having two variables “head” and “tail”, we can make insertion and removal on the list ends have a run time of O(1). For insertion and removal operations that happen somewhere in the middle of this list, this still requires an O(N) algorithm.

Circular Doubly Linked Lists

In a Circular Doubly Linked List, the first node's previous field (typically set to null) is set to the last node. Likewise, the last node's next field (typically null) is set to the first node. The result is a list with no definitive head or tail.

Node Lists

- A variation on Linked Lists designed for speed

Node.java

```
class Node<T>{
	private Node<T> prev;
	private Node<T> next;
	private T data;

	public Node(Node<T> prev, Node<T> next, T data){
		this.prev = prev;
		this.next = next;
		this.data = data;
	}

	public T get(){
		if( prev == null && next == null ){
			System.out.println( "Error: element does not exit." );
			System.exit(1);
		}
		return data;
	}
	public Node<T> getNext(){
		return next;
	}
	public Node<T> getPrev(){
		return prev;
	}
	public void setNext(Node<T> next){
		this.next = next;
	}
```

```
	public void SetPrev(Node<T> prev){
		this.prev = prev;
	}
	public void set(T data){
		this.data = data;
	}
}
```

Node.java

```
class NodeList<T>{
	private Node<T> head;
	private Node<T> tail;
	private int size;

	public NodeList(){
		size = 0;
		head = new Node<T>(null, null, null);
		tail = new Node<T>(null, null, null);
		tail.setPrev(head);
		head.setNext(tail);
	}

	public Node<T> first(){
		return head;
	}
	public Node<T> last(){
		return tail;
	}
	public int size(){
		return size;
	}
	public int isEmpty{
		return size == 0;
	}
	public Node<T> prev(Node<T> p){
		return p.getPrev();
	}
	public Node<T> next(Node<T> n){
		return n.getNext();
	}

	//update methods

	//set -> sets data in a node
	//addFirst -> sets a new node at the beginning
		// of the list. the user is responsible for making their own node
	//addLast -> sets new node as last element
	//addBefore -> takes two arguments: a node in the list and a node not
```

```
        // in the list.  The node not in the list is added before the node
        // in the list.
    //addAfter -> takes two arguments: a node in the list and a node not
        // in the list.  The node not in the list is added after the node
        // in the list.
    //remove -> remove a node
```

Array Lists

- An array list is also known as a vector
- whereas Linked Lists are memory efficient, they are slow
- An array list is fast, but has poor memory usage.
- An array list is implemented with a standard array

Theory Behind and Array List

- The perceived size to the user
- The actual size known to the data structure

Three Fields

- size (perceived size)(starts at 0)
- real-size (actual size of an internal array)(starts at 2)
- an array of elements

ArrayList.java

```
class ArrayList<T>{
    private int size;
    private int real_size;
    private T[] elements;

    public ArrayList(){
        size = 0;
        real_size = 2;
        elements = (T[])new Object[real_size];
    }

    private void resizeArray(){
        real_size *= 2;
        T[] newArray = (T[])new Object[real_size];
        for( int i = 0; i < elements.length; i++ ){
            newArray[i] = elements [i];
        }
        elements = newArray;
    }
    public boolean isEmpty(){
        return size == 0;
```

```
	}
	public int size(){
		return size;
	}
	public T get(int pos){
		if( pos < 0 || pos >= size ){
			System.out.println( "Error: Array Out Of Bounds: " + pos );
			System.exit(1);
		}
		return elements[pos];
	}
	public void set(int pos, T data){
		if( pos < 0 || pos >= size ){
			System.out.println( "Error: Array Out Of Bounds: " + pos );
			System.exit(1);
		}
		elements[pos] = data;
	}
	public T remove(int pos){
		if( pos < 0 || pos >= size ){
			System.out.println( "Error: Array Out Of Bounds: " + pos );
			System.exit(1);
		}
		T temp = elements[pos];
		for( int i = pos; i < size-1; i++ ){
			elements[i] = elements[i+1];
		}
		size--;
		return temp;
	}
	public void insert(T data, int pos){
		if( pos < 0 || pos > size ){
			System.out.println( "Error: Array Out Of Bounds: " + pos );
			System.exit(1);
		}
		if( size = real_size ){
			resizeArray();
		}
		for( int i = size-1; i >= pos; i-- ){
			elements[i+1] = elements[i];
		}
		elements[pos] = data;
		size++;
	}
}
```

CSCI 112
Tuesday, July 27, 2010

Concurrency

Concurrency – two or more events happening at the same time.

Two Types of Processes

- **Process**
  - has memory
  - has a *process identifier* (PID) in the operating system
- **Thread**
  - a "lightweight process"
  - does not have a PID
  - has a parent process

Forking a Process

The is a Unix term. This means a process can/will duplicate itself during runtime.

Java Uses Threads

By default, there are two threads in every Java program:
- Main execution thread
- Garbage Collector

MyThread.java

```
import java.util.*;

public class MyThread implements Runnable{
	public void run(){
		System.out.println( "Hello, World!" );
	}
	public static void main( String[] args ){
		MyThread a = new MyThread();

		Thread t = new Thread(a);
		//Thread constructor requires any object that implements runnable

		t.start(); //begin the new thread
	}
}
```