Software Systems (COE 318): Notes

Adam Szava

Fall 2021

Introduction

This is my compilation of notes from Software Systems (COE 318) from Ryerson University. All information comes from my professor's lectures, the textbook *Head First Java*, and online resources.

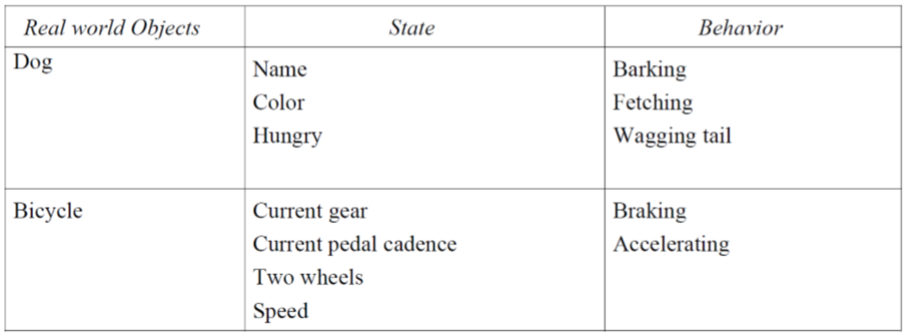
Week 1: Software Development Cycle

This course focuses on the high-level programming language *Java*.

This language is called *object-oriented*, meaning you define objects first, then give the objects abilities, and then link the objects together. All code is run within objects in Java.

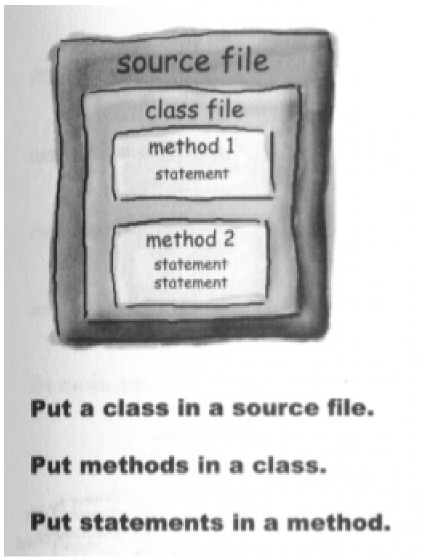
A **class** is a template for an object. Take for example the Dog class, all dogs have: names, colors, weights, etc. An object is a specific dog, which a specific name, color, weight, etc. Sometimes we call the specific object of a class, and *instance* of a class.

An object is an instance of a class, all instances of a class share behaviours, and states. As in all dogs share the ability to wag their tail, eat, run, and they also all share the states of sleep, awake, hungry, in which they may do different things.



Real world Objects	*State*	*Behavior*
Dog	Name Color Hungry	Barking Fetching Wagging tail
Bicycle	Current gear Current pedal cadence Two wheels Speed	Braking Accelerating

In general a Java document is structured like the following:



Put a class in a source file.

Put methods in a class.

Put statements in a method.

Hello World! and Basic Syntax

The following is the code for Hello World!:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
         System.out.println("Hello World!");
   }
}
```

The first line opens up a general class called *HelloWorld*, remember all code must be run in a class, and so the class which the whole file runs in is the same as the name of the file.

Then we open a new class called the *main* class which then runs the code to print.

Print Statements

```
System.out.print(”string”);
```

This statement prints out the string in the command prompt.

Println Statements

$\boxed{\text{System.out.println("string");}}$

This statement prints out the string in the command prompt, and adds a line break after.

Notice that all statements end in a semicolon.

Variables

A variable must be declared before use with its data type. A variable is a container which can hold certain content.

Variable Declaration

$\boxed{\text{type name;}}$

This statement declared a variable of type *type*, and of name *name*.

The **assignment** operator assigns a value to a variable, like *C*, the assignment operator is =.

Variable Assignment

$\boxed{\text{varName = dataOfCorrectType;}}$

$\boxed{\text{dataType varName = dataOfCorrectType;}}$

Comments

Comments are done either by putting // at the beginning of a line, or enclosing multiple lines in /* · · · * /.

Methods

A method is a unit of code which can do work. You call the method when you want to do that unit of work. A *method* is a function that is part of class. Since all code is part of a class in Java, then all functions are called *methods*.

Methods have return types and parameters. The return type is what data type the function will return, and the parameters are what is going into the function.

Declaring Methods

Within a class you can declare a method by the following syntax:

```
returnType methodName(dataType1 para1, dataType2 para2, ..., dataTypeN, paraN) {
        statements;
        statements;

        return dataOfCorrectType;

}
```

... and calling methods in the main file would be just like you call functions:

$$\boxed{\text{methodName(arg1, arg2, ..., argN);}}$$

Classes

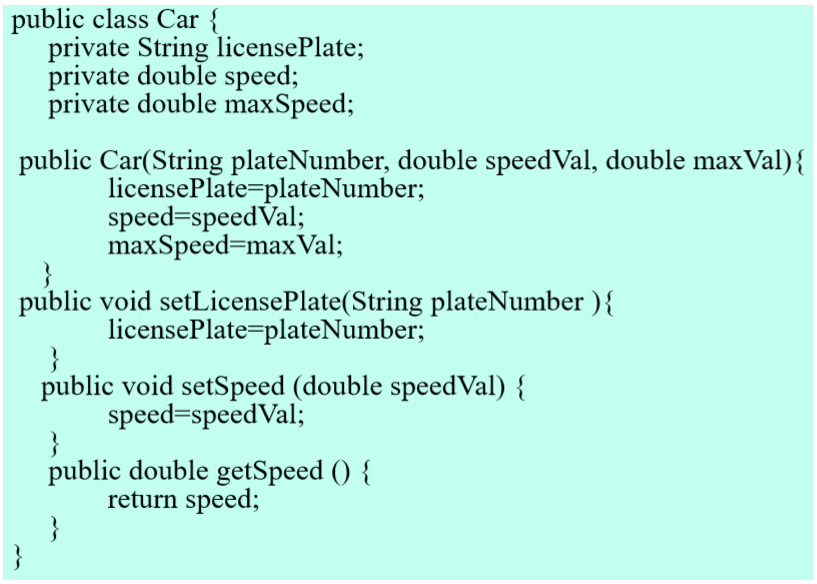
A class is a template which describes many objects. A class contains variables and methods which all instances of the class will have their unique versions of. You can declare a class like the following:

```
public class className {
        // declare variables
        //(these are called instance variables)
        private dataType instVar1;
        // there can be as many instance variables as you want

        //declare constructor
        public className(dataType1 para1, dataType2 para2, ..., dataTypeN, paraN) {
        }

        //declare methods
        returnType methodName1(dataType1 para1, dataType2 para2, ..., dataTypeN, paraN) {
                statements;
                statements;
        return dataOfCorrectType;
        }
        // there can be as many methods as you want
}
```

The use of public and private will be explained in later weeks.

The first method of the same name of the class is called the *constructor*. This method creates an instance of the class when called, as in an object that follows the template. The arguments are the initial values that the instance of the object should have. The constructor is called like any method. The job of the constructor is to assign the object's instance variables to the variables that were just input in the call.

The following is an example of a fully worked class called Car with variables and methods:

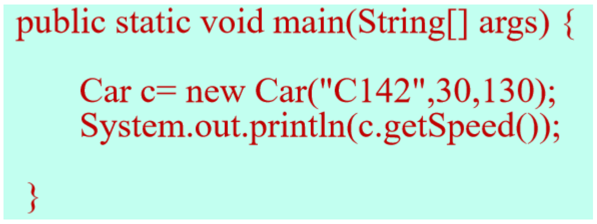


```
public class Car {
    private String licensePlate;
    private double speed;
    private double maxSpeed;

 public Car(String plateNumber, double speedVal, double maxVal){
          licensePlate=plateNumber;
          speed=speedVal;
          maxSpeed=maxVal;
    }
 public void setLicensePlate(String plateNumber ){
          licensePlate=plateNumber;
    }
   public void setSpeed (double speedVal) {
          speed=speedVal;
    }
    public double getSpeed () {
          return speed;
    }
}
```

Objects

We can declare objects after creating the class to which they belong. We do this in a separate method called the *main* method.

Given the previous example, the following code would be valid:



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

      Car c= new Car("C142",30,130);
      System.out.println(c.getSpeed());

}
```

Here, in the main method, a new variable c is declared to be of class Car. We use the keyword *new* to trigger the constructor and we then call the constructor with the initial values.

In general that is:

Creating an Object of a Class

```
className varName = new className(arg1, arg2, ..., argN);
```

We then use the dot operator to access the getSpeed method that is within the object c. c has this object since it is of the class Car. The dot operator works for methods and public variables, however it is bad form to access variables like that and later in this text we will describe a better way to access and change variables.

Week 2: Programming Languages

Java has a list of *primitive data types*:

- **byte**: represents one signed byte as a unit of 8 bits, meaning the numbers from $(-127, 127)$.
- **char**: represents 2 unsigned bytes, unicode from 0 to 65535.
- **short**: represents 2 signed bytes, numbers from $(-32768, 32768)$.
- **int**: represents 4 signed bytes, meaning numbers from $(-2147483648, 2147483648)$.
- **long**: represents 8 signed bytes.
- **float**: represents 4 signed bytes.
- **double**: represents 8 signed bytes.

Java also has a list of basic arithmetic operations:

- Addition $+$.
- Subtraction $-$.
- Multiplication *.
- Division $/$.
- Remainder or Modulus %.

We say that the $+$ operation is *overloaded*. An overloaded operation means it changes its function depending on the inputs. For example in Java $5+2$ will evaluate to 7, but *"Hello" + "Friend"* will evaluate to *"HelloFriend"*. This is called *concatenation*. In general, Java will add as expected from left to right until it reaches a string, then everything is converted to a string from then on. For example:

$$2 + 5 + \text{"Wow"} + 4 \implies \text{"7Wow4"}$$

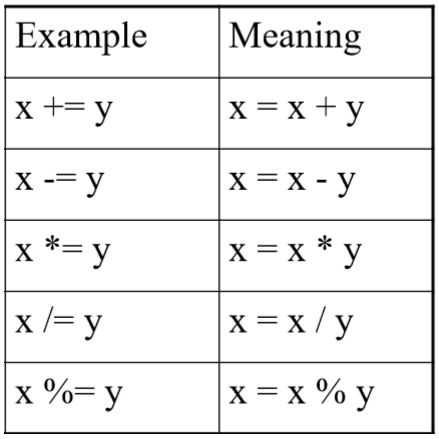
Java also has equality operators:

- $==$ returns true if both operands are equal.
- $!=$ returns true if both operands are not equal.

Java also has relational operators:

- Greater than: $>$.
- Less than: $<$.
- Greater than or equal to: $>=$.
- Less than or equal to: $<=$.

Java also has compound operators:



Example	Meaning
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Java also has the increment and decrement operators:

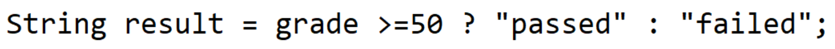
- $++i$: increment i and then use the new value to evaluate the expression.
- $i++$: use the current value of i to evaluate the expression, and then increment i.
- $--i$: decrement i and then use the new value to evaluate the expression.
- $i--$: use the current value of i to evaluate the expression, and then decrement i.

If statements, else if statements, switch statements, while loops, do while loops, and for loops are exactly the same as C even in syntax.

There is a unique syntax for a simple if statement, as follows:

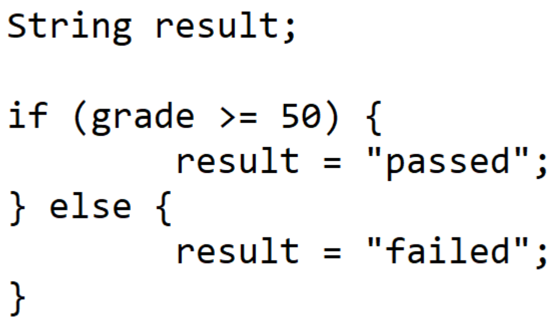
$$\boxed{\text{dataType varName} = \text{condition ? ifAssignment : elseAssignment}}$$

Here is an example to show the meaning of the code... the following code:



```
String result = grade >=50 ? "passed" : "failed";
```

... is equivalent to this code:



```
String result;

if (grade >= 50) {
        result = "passed";
} else {
        result = "failed";
}
```

Objects and Classes

When you declare a variable to be of a certain class, it is instantiated to be *null*, meaning it does not refer to any particular object yet. When you use the *new* keyword memory is allocated within the computer in the format of the class.

Typically, the main method is in a different class called the *Helper* or *Tester* class, which *ONLY* contains the main method and nothing else. This helper class should be in the same project as the defined classes.

Accessing Data from Objects

We introduced the dot notation to access variables or methods of an object last week. Say you have some object *miniVan* of class Car, and you want to access/change the speed of that car. This object would thus need a *speed* variable. You can access that variable using the following notation:

miniVan.speed;

You could use this to both read the value, but also write to the value:

miniVan.speed = 5;

This is a security problem, maybe you shouldn't just be able to see the value from any method (like if it were a password) or maybe you shouldn't be able to set the value to whatever you want (like if its the speed of a remote control car). This is why we use *getter* and *setter* methods.

Getter and Setter Methods

A getter method is a method which returns the value of a variable within an object. A setter method is a method which sets the value of a variable within an object.

Getter Methods

Say you have some object with instance variable declared as:

dataType instVar1;

Then its getter method would be:

```
dataType getInstVar1(){
        return instVar1;
}
```

Setter Methods

Say you have some object with instance variable declared as:

dataType instVar1;

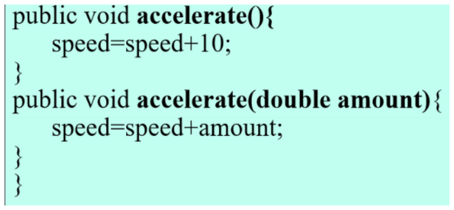
Then its setter method would be:

```
void setInstVar1(dataType input) {
        instVar1 = input;
}
```

Week 3: Variables

Method overloading is when you give different function definitions for the same function depending on the number and type of parameters (called the signatures).

In the following example:



```
public void accelerate(){
    speed=speed+10;
}
public void accelerate(double amount){
    speed=speed+amount;
}
}
```

You can see that accelerate is defined twice, once for no input, and once for an input of type double.

Many methods built into Java are overloaded like the print function which can take in many different types of inputs.

When you overload a method you give it the same name, but different parameters and/or return types.

Mutable versus Immutable

A mutable class is a class which contains setter methods, as in the object of the class can be mutated after its creation. An immutable class has no setter methods, and so the only way to set the values of the instance variables of the class is to set it when the constructor is called.

Constructors

Constructors are the method which is called to create an instance of the class. Constructors always have the same name as the class, and do not have a return type.

You technically don't need to create a constructor explicitly, what happens is when you create a class with no constructor, a default constructor is created which has no code. You *need* a constructor if you want to assign values to your instance of a class. Constructors can also be overloaded in the same way as methods to allow for different objects of the class to be created with different creation calls.

this Keyword

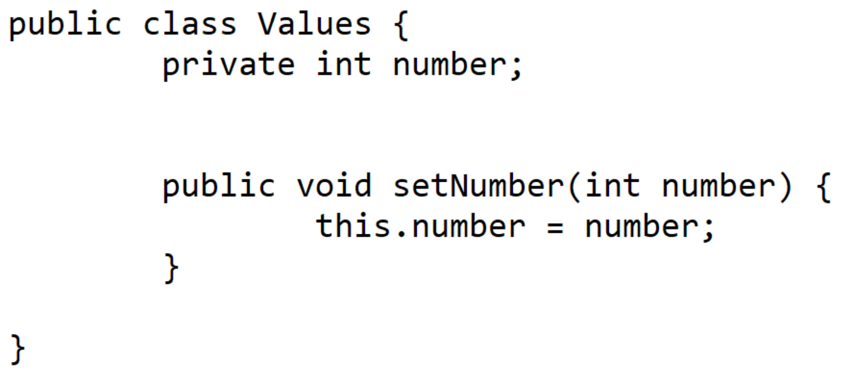
Say you have a method which has an input variable *number* and you want to assign that input to an instance variable of the object also called *number*. You are **not** allowed to write:

number = number

What you have to use is:

this.number = number

Which means the *number* variable assigned to this object, should be assigned to the input variable *number*... for example:



```
public class Values {
        private int number;


        public void setNumber(int number) {
                this.number = number;
        }

}
```

We say that *this* is a reference to the invoking object of the method.

Types of Variables

We can classify variables based on type:

1. Primitives (int, short, etc.)
2. Reference (Any data of type object)

... or by usage:

1. Instance variables.
2. Parameters (of methods).

3. Local variables (which are instantiated and only used within a method).

When you declare a class, it becomes a new data type.

Boolean is a data type which can either be *true* or *false*.

Conversion Between Primitive Types

The order of *sizes* of primitive types is:

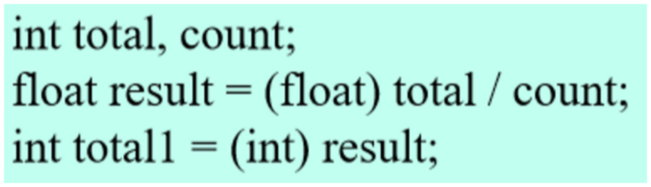
byte ¡ short ¡ int ¡ long ¡ float ¡ double

There are two types of conversions of primitives:

1. Widening conversions, which go from small data types ot larger ones.

2. Narrowing conversions, which go from large data types to smaller ones.

Both types can be completed by *type casting*. For example:



```
int total, count;
float result = (float) total / count;
int total1 = (int) result;
```

It is necessary to cast the quotient of integers into a float.

Variable Initialization

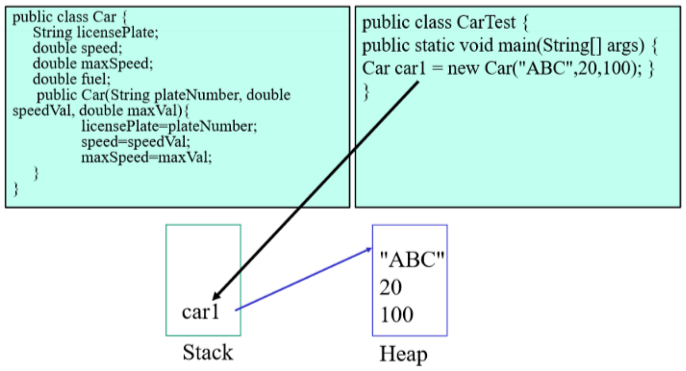
All variables of number types are instantiated to 0 while char and null get the null character. The null type is a data type which can reference any type of object.

Data Storage in Memory

Data is stored in the memory of the computer. Primitive data types all have their owe specified size, while reference variables require 4 bytes plus the memory required for the instance variables of the object being referenced.

Memory is separated into two distinct sections called the heap and the stack. The heap collects any memory required by the reference to an object, and to its variables, while the stack holds local variables and parameters.

The stack can grow and shrink dynamically.



```
public class Car {
    String licensePlate;
    double speed;
    double maxSpeed;
    double fuel;
     public Car(String plateNumber, double
speedVal, double maxVal){
            licensePlate=plateNumber;
            speed=speedVal;
            maxSpeed=maxVal;
    }
}
```

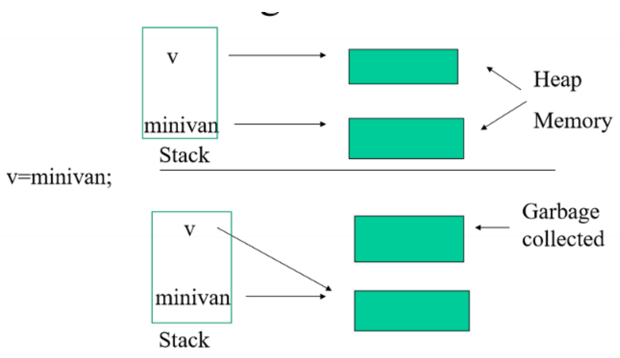
```
public class CarTest {
public static void main(String[] args) {
Car car1 = new Car("ABC",20,100); }
}
```

When a method is invoked, its parameters are on the stack, then additional space is allocated for all local variables in the method. When the method is complete, the space on the stack is freed from the local variables.

Managing heap memory is done differently. Java keeps track of how many times in the code a specific variables point to the data held by an object in the heap. If that reference count drops to zero at some point in the code, then the object will never be referenced again, and so it can be deleted. These objects with a reference count of zero are referred to as *orphaned* and the deletion process is called *garbage collection* by the *garbage collector.*

If all this data deletion didn't happen, then big programs would eat all the computer's RAM until it crashes.

If you assign one reference variable to another, then whatever was originally being referenced to by the original reference variable is garbage collected, and now both reference variables point to the same data in the heap.



Arrays

An array can store multiple values of the same type, including objects of a class. Indexing begins at 0 in Java.

Declaring Arrays

dataType [] arrName;

arrName = new dataType[arrLength]

... the first line can also be written as:

dataType arrName[];

... or alternatively all in one line:

dataType [] arrName = new dataType[arrLength]

The index of an array must specify a valid element, as in be between 0 and $arrLength - 1$ inclusive.

Arrays are objects which have built in methods, such as the length method.

Array Length

$$arrName.length;$$

Returns an integer equal to *arrLength*.

When arrays of elements are created, you need to initialize every element of the array, they will not be initialized automatically. To begin, every element contains the null character.

Week 4: Using Classes and Objects

The String Class

Strings are built in classes in Java. There are two ways to declare a string:

- **String Object**:

  String strName;

  strName = new String(”strContents”)

- **String Literal**:

  String strName = ”strContents”

The **string pool** is a large section of memory for strings declared **as a literal**. Here, strings are held only once, meaning if you have multiple variables assigned to same string, they all point to the same single string in the memory.

There are same specific methods built into all strings:

Character at Index

The following method returns the character or type char at index i of the string:

strName.chatAt(i)

Concatenate

The following method concatenates the string *strName* with the string *strConc*. It returns a new string. Recall concatenation just mean connect to the end of:

strName.concat("strConc")

Length of String

The following method returns an integer equal to the length of the string.

strName.length()

Replace Characters in a String

The following method replaces all instances of character char1 with char2, and returns a new string object.

strName.replace(char1,char2)

Note that char1 and char2 should have single quotes around them each.

Substrings

The following method returns the string between the ith index and the end of the string:

strName.(i)

The following method returns the string between the ith and jth index of the string:

strName.(i,j)

These methods can read up to the length of the string.

Comparing Variables

Comparing floats cannot be done directly because of inaccuracies in the memory. For this reason a good way to compare floats is using a tolerance (how inaccurate you are willing to be):

```
if(Math.abs(a-b) < TOLERANCE) {
        // statements
}
```

Essentially this code is asking if the difference between the floats is less than some tolerance decided by the program.

Comparing chars can be done using the equality operator or the comparison operators since all chars have an ASCII number. The following table describes this:

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing objects is more complicated. If you want to compare strings, you can use the equality operator but it will not work as you expect. *The equality operator between strings returns true if both reference variables reference the same memory in the heap.*

If you declare two different string variables to reference the same string, then their equality operator will return true. Otherwise you need to use the following syntax:

Equals Method for Strings

The following will return true if the two strings are equal regardless of how they were declared:

strName1.equals(strName2)

If you pass a primitive data type into a method, changing the value of the variable in the method does not change the value of the variable outside the function as a copy of the input variable is created as a local variable.

If you pass an object into a method, you are just passing the reference to the memory location, so changing an object inside of a method *does* change the object outside of the method.

Linking Objects

Objects can be linked together by containing a reference to another instance of the class as one of their instance variables. We say then that the two objects are linked. In the instance variable declaration section of the class there would be a declaration line of code like the following:

Class linkedObjectRef;

Week 5: Writing Classes

Packages

Packages are used to group together similar classes. This is what real software engineers are putting together – packages of classes which serve some purpose to a main file.

Say you want to use a class you wrote for another project, in a new project. Then you would have to import that class using the packages syntax. The following code declared all the classes of the file to be part of a package:

package packageName;

... now this code imports that package class to another Java document:

import packageName.className

... or to import **all** the classes:

import packageName.*

Encapsulation

Encapsulation is the idea that certain variables should not be able to be edited by every class. This is why we use setter and getter methods, to promote *encapsulation.* This can be accomplished by *visibility modifiers* which include:

1. *public*: methods and instance variables declared with this modifier can be accessed without getter and setter methods.
2. *private*: methods and instance variables declared with this modifier cannot be accessed without getter and setter methods.

If no modifier is declared, then by default the variable or method can be accessed by any class in the same package.

Generally there are two good rules to go by:

1. Instance variables should be declared as private.
2. Methods that provide the object's services should be declared as public. These methods are also called *service methods.*

A private method that is created to assist the service method is called a *support method.*

Accessors and Mutators

Formally, the getter method is referred to as an accessor, and the setter method is referred to as a mutator.

Scope

Scope of a variable is the section of code that has access to the memory referred to by that variable name. Instance variables have class scope, meaning any method in the class has access to it. Local variables have block scope meaning they can only be accessed by the block of code they were declared in.

Arrays and 2D Arrays

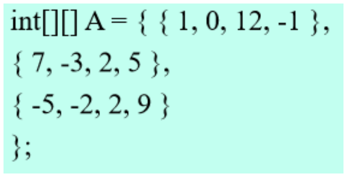
If you pass an array into function, it will be passed by reference and so changing it inside the method will change it outside the method. The same is **not** true for particular indexes of the array.

In Java 2D arrays are just arrays of arrays. This is like the *computer scientist method* in CPS 125.

When you declare an $m \times n$ 2D array:

```
dataType[m][n] arrName;
```

Every one of the m elements of the first array is a reference to an array of length n. Arrays can also be declared like the following:



```
int[][] A = { { 1, 0, 12, -1 },
{ 7, -3, 2, 5 },
{ -5, -2, 2, 9 }
};
```

User Input

Java has a built in class called *Scanner* which is part of the *java.util* package. To activate this class and package, you need to do the following:

- Read from keyboard:

```
import java.util.Scanner;

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

- Read from file:

```
import java.util.Scanner;

Scanner s = new Scanner(new FileReader(”myFile”));
```

To assign a string to a user input, use the following input:

User Input

The following code waits for user input, then assigns it to the string:

$$strVar = scan.nextLine();$$

The scanner method breaks the input into *tokens* which are separated by spaces. Essentially, words numbers, and symbols are tokens.

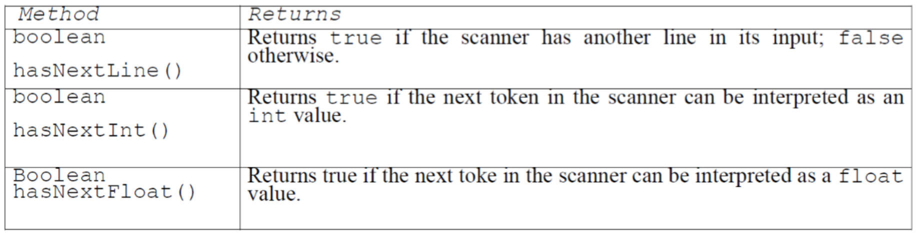
To just scan the next token in a string, use the following line:

strVar = scan.next();

Or to read until the next integer in the input string:

intVar = scan.nextInt();

The following are a few more methods which are useful when using the Scanner class:



Method	*Returns*
`boolean hasNextLine()`	Returns `true` if the scanner has another line in its input; `false` otherwise.
`boolean hasNextInt()`	Returns `true` if the next token in the scanner can be interpreted as an `int` value.
`Boolean hasNextFloat()`	Returns true if the next toke in the scanner can be interpreted as a `float` value.

Week 6: Implementation of Classes

Enhanced For Loops

Enhanced for loops are a quick way to write for loops in Java when using an array. Essentially it is when you want to say: "*for each element in this array...*". Say you have some array *arrName* of type *dataType*, then if you want a for loop to access each element:

for (int element : arrName) {. . . }

Within the for loop, the current element of the list can be accessed with the variable *element.*

Object Class

All classes are implicitly derived from the object class. When we learn about inheritance the object class is the parent of all classes.

Object reference variables can be typecast into referencing others as well.

There are two types of object casting:

1. **Upcast:** This is where we assign references to parent objects, and the references point to children objects. For example if we define some object *st* of class *Student*, and some object *obj* of class *Object*, then the following is an upcast:

   Object obj = st;

2. **Downcast:** This is where we assign reference to children objects, and the references point to parent objects. For example, the same example as above but as a downcast:

   Student st = (Student) obj

   Downcasts need explicit typecasting.

ArrayList

An *ArrayList* is an object built into Java which acts like a more advanced array. Essentially it is dynamically allocated, compared to regular arrays which need their size defined immediately. Like any class from an external package the class needs to be imported:

import java.util.ArrayList;

... and then an array list needs to be declared:

Arraylist arrListName = new ArrayList()

Array lists have a lot of functionality built into them, for example:

- *arrListName.add(input)*: this adds the *input* to the end of the list, input can be of any data type.
- *arrListName.indexOf(entry)*: this returns an int which is the index of *entry* in the list.
- *arrListName.remove(index)*: this removes the list entry at index *location* **AND** shifts all later entries down an index.
- *arrListName.add(index, input)*: this first of all shifts all entries from *index* onward up a place, leaving *index* empty, and then assigns that place in the list as *input.*
- *arrListName.size()*: this returns an integer equal to the number of entries in the array list.

Array lists can take any combination of types, but you can force it to only accept one type of data type (*dataType*) in its declaration:

ArrayList <dataType> myList = new ArrayList <dataType> ();

Command Line Arguments

The main function takes in an array of strings called command line arguments. The main function is always declared:

public static void main(String[] args)

... and you can access the strings by using the *args* variable.

Variable Length Parameter List

A variable length parameter list is a way for a function to take in a different number of inputs without having to overload the method for every number of inputs. Say for example you want to make an averaging method, then you would need to overload program it for 2 inputs, then 3, then 4, and so on.

Variable length parameter lists essentially let you input an array of number without declaring it as that so that the input can be however long you want it to be.

In the method definition, the final input variable of any type can be a variable length parameter, denoted with $dataType \ldots varName$.

Static Variables and Methods

A variable or method declared as *static* as a modifier will be set throughout the instances of the class, without needing to even be initialized once. A static variable exists as part of the class. Static variables and methods can be invoked without an instance of the class. Notice how main is always static since Java needs to be able to enter the main method without the main class even existing yet.

Something to keep in mind is that static variables and methods have no access to non-static variables and methods without initialization because the variables simply don't exist otherwise.

Static variables are useful when a variable should be global among all instances of a class. Like the count of the instances of that class. Static methods are useful when the method only depends on the parameters.

The Final Keyword

When a variable is declared as static it cannot be changed at a later time. There are two ways to instantiate a final variable:

1. Right away at declaration in the instance variables.
2. In the constructor.

There are no other options, if you fail to do this Java will throw an error.

For security purposes, as in to make sure data isn't randomly lost, final variables can be used as parameters to a function.

Static final variables are used when the variable should be the same across all classes, and never be changes. This works for numbers like PI.

Order of Modifiers

We have learned about a lot of modifiers. The order of the modifiers doesn't matter as long as the data type is last, however what is recommended is something like the following:

private static final double varName;

Week 7: Testing using JUnit

Software testing is the process in which you run code and compare its result with what is expected, in order to ensure the function of the code.

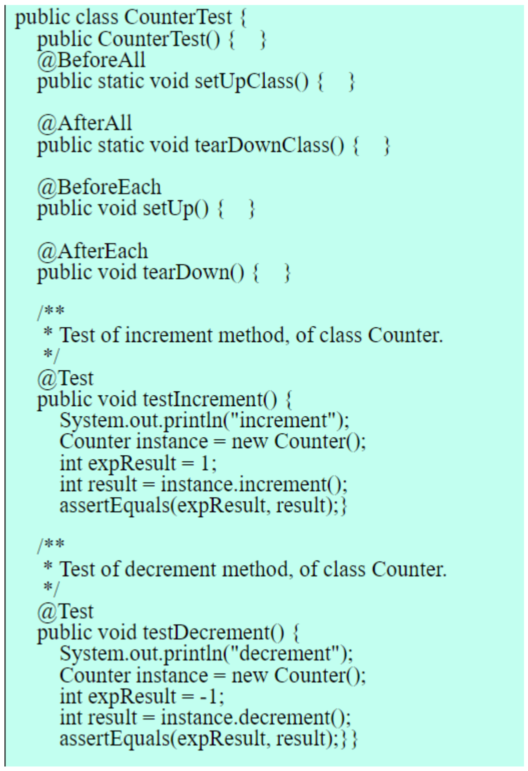
A test case tests the response of a method to some particular inputs. **Unit Testing** is a test of a single class. These tests usually test each component of the class to see if they behave as expected. **Integration Testing** is a test of how well classes work together.

Java has a tool to do unit testing called *JUnit*, and it is integrated into NetBeans.

To use JUnit, you typically create a new class, for example if we want to test the following code:

```
public class Counter {
        int count = 0;
        public int increment () {
                return ++count;
        }
        public int decrement() {
                return --count;
        }
}
```

Then we would use the following class:



```
public class CounterTest {
    public CounterTest() {    }
    @BeforeAll
    public static void setUpClass() {    }

    @AfterAll
    public static void tearDownClass() {    }

    @BeforeEach
    public void setUp() {    }

    @AfterEach
    public void tearDown() {    }

    /**
     * Test of increment method, of class Counter.
     */
    @Test
    public void testIncrement() {
        System.out.println("increment");
        Counter instance = new Counter();
        int expResult = 1;
        int result = instance.increment();
        assertEquals(expResult, result);}

    /**
     * Test of decrement method, of class Counter.
     */
    @Test
    public void testDecrement() {
        System.out.println("decrement");
        Counter instance = new Counter();
        int expResult = -1;
        int result = instance.decrement();
        assertEquals(expResult, result);}}
```

There are many parts to this code:

- Name the class the same as the code but with Test after it.
- Empty constructor.
- Lines that begin with @ are *annotations*.
- @BeforeAll and @AfterAll are methods which run once per class.
- @BeforeEach and @AfterEach are methods which run once per test.
- *public void setUp()* creates a test by creating and intializing objects and values.
- *public void tearDown()* releases any system resources used by the test.
- In the Test methods, we make a new instance of the class we are testing and then write the expected result. We then call the function and...
- Compare the expected result with the actual result using the assertEquals(expResult, result);

Within a test, you want to call the method being tested, get the result, and assert what the correct result should be.

Assertions include:

- *assertTrue(boolean test)* asserts that the condition is true.
- *assertFalse(boolean test)* asserts that the condition is false.
- *assertEquals(expected, actual)* asserts that the two inputs are equal, they can be any primitive of the same type or objects.

You can also fail a test by calling the fail method:

- *fail()* fails the test without message.
- *fail(String failResponse)* fails the test with the string given.

Week 7-9: Inheritance

The main idea of inheritcance is that some objects behave similarly. For example in a game you want the player to not be able to walk through fences, doors, walls, trees. It would be ineffient to check each individually, so you can create a *parent class* which they all belong to called *stoppingObjects* and the code for the player just needs to check if they are running into the parent object.

You want to put common code in a super class (also called a parent class), each sub-class (also called child class) inherits from the super class, and can reuse code written for the superclass. A subclass can inherit methods and instance variables.

In the subclass, you can add new methods and variables that are unique to the subclass while still inheriting shared code. Additionally, a subclass can overwrite inherited code if needed.

Let's say you wanted to have a Car class and a Truck class as follows:

```
public class Car{
        String licensePlate;
        String status;
        int numWheels=4;
        int tankSize=65;
        public String getPlate(){}
        public void reFuel(){}
        public String getStatus(){}
}


public class Truck{
        String licensePlate;
        String status;
        int numWheels=18;
        int tankSize=200;
        public String getPlate(){}
        public void reFuel(){}
        public String getStatus(){}
}
```

There is obviously a lot of similarity between the two, and so we can reduce this reuse of code by calling them both children of the following parent class:

```
public class Vehicle{
        String licensePlate;
        String status;
        int numWheels;
        int tankSize;
        public String getPlate(){return licensePlate;}
        public void reFuel(){
                System.out.println("Override this!");}
        public String getStatus(){return status;}
}
```

We can then re declare the two classes in the following way:

```
public class Car extends Vehicle{
        int numWheels=4;
        int tankSize=65;
        @Override
        public void reFuel(){
                System.out.println("Fuel must be gasoline");}
}

public class Truck extends Vehicle{
        int numWheels=18;
        int tankSize=200;
        @Override
        public void reFuel(){
                System.out.println("Fuel must be diesel");}
}
```

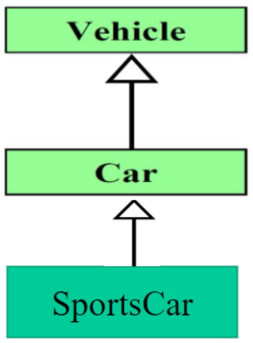
Notice the use of the word **extends** and the @Override. The override is optional but it makes Java check if you are actually overriding something so it is good form.

When you call a method in a class, the most specific form of it is called. If the class itself does not have a definition for the method, the version of it in the nearest parent class is called.

Only public and protected methods can be overridden, private methods cannot as they only are accessible by the class it is declared in. Additionally, if the parent class method is declared as public, then the child class method by the same name cannot be private or protected, it can only be as restrictive or less than the parent.

The *protected* access modifier affects instance variables by making it so only subclasses of the class have access to information, and no other classes. It acts as a modifier midway between public and private.

The chain of parent and child classes can go up to 60 levels deep.



Whenever you instantiate a subclass, the default constructor of the super class is called implicitly. If you however only have non-default constructors in the parent class, then it must be called explicitly or else Java will throw an error.

The *super* keyword is used to call the constructor of the super class, and must be the first statement in the constructor of the subclass. The following is an example of its use:

```
public class Car extends Vehicle{
        public Car(String licensePlate){
                super(licensePlate); //must be first line!
        numWheels=4;
        tankSize=65;
        }
        public void reFuel(){
                System.out.println("Fuel must be gasoline");}
}
```

Even without a call to super, the default constructor will be called. You can also use the *super* keyword to access methods from the super class, such as (in the Car class):

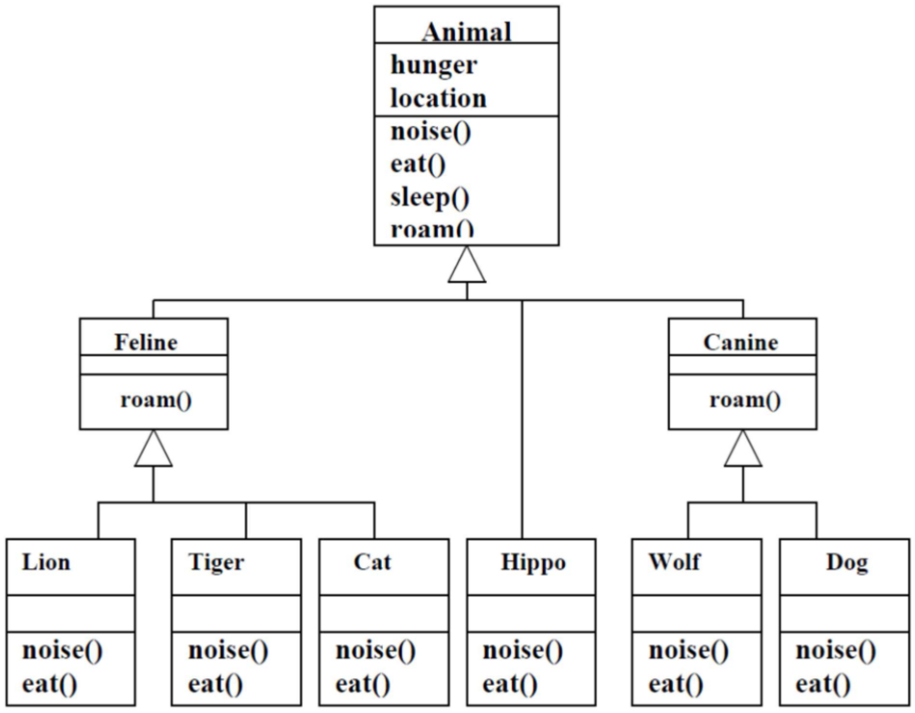
```
public String getPlate(){
        return "The car's plate: "+ super.getPlate();
}
```

Super can only go to the nearest parent class, and not to higher degree parent classes.

Class Hierarchies

A class hierarchy is a way to organize common characteristics of classes. You want to look for objects which have common states and behaviours, and design a class which represents this common state and behaviour.

The following is an example of a class hierarchy diagram:



Final Classes

A class declared as *final* cannot have any subclasses. Similarly, a *final* method cannot be overwritten. For example:

```
final class EndOfTheLine {
        public void aFunction() {
                System.out.println( "Hi Mom" );
        }
}
class ThisWillNotWork extends EndOfTheLine { //Compile Error
        public void aFunction(){
                System.out.println("Overriding");
        }
}
```

Abstract Classes

An abstract class which is only to be used as a parent to other classes, and no objects can be created of an abstract class. For example if you have a Vehicle class, and then a Car and Truck class, you only want to be able to make instances of the Car and Truck classes, not directly of the Vehicle class.

An abstract class is a list of generic behaviours, not an object itself.

An abstract class has no use unless it is extended.

Abstract methods are method that only an abstract class can have. They have no body and **must** be overridden by the subclasses.

The following is an example of an abstract class:

```
public abstract class Vehicle{
        private String licensePlate;
        private String status;
        protected int numWheels;
        protected int tankSize;
        public Vehicle()
                {System.out.println("In Vehicle's def. constructor");}
        public Vehicle(String licensePlate)
                {this.licensePlate=licensePlate;}
        public String getPlate()
                {return licensePlate;}
        protected abstract void reFuel();//can't have a body...
        // ...otherwise compile error!
        public String getStatus()
                {return status;}
}
```

Interface

An interface is a way for a class to inherit from multiple parents. You can only every be a subclass, of one parent class, but you can *implement multiple interfaces.*

An interface is **not** a class although they look similar, an interface is a collection of methods which any class that implements the interface must implement. An interface defines a protocol of behaviour that can be implemented by any class.

An interface has only abstract classes, and so it cannot implement any code, whereas a abstract class can implement some code. A class can implement many interfaces but can have only one superclass.

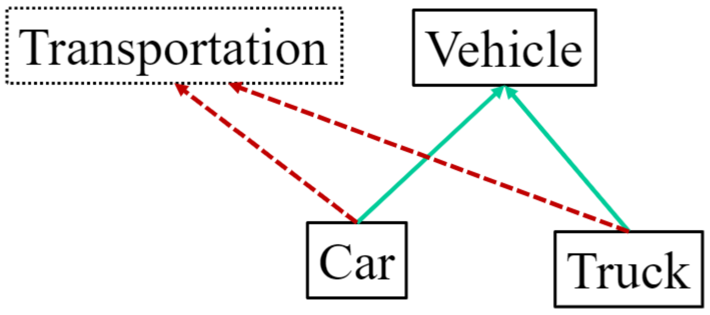
The following is an example of an interface:

```
public interface Transportation{
        public abstract void safetyCheck();
}
```

Then you would add the interface to the car class in the following way:

```
public class Car extends Vehicle implements Transportation{
        public Car(String licensePlate){
                super(licensePlate);
                numWheels=4;
                tankSize=65;
        }
        public void reFuel(){
                System.out.println("Fuel must be gasoline");}
        public void safetyCheck(){
                System.out.println("Car's safety check");}
}
```

To understand the relationship between classes, abstract classes, and interfaces, often the following diagrams are used to visualize:



The Car and Truck classes are children of the Vehicle class, and they both implement the Transportation interface. More on these diagrams later.

In an interface all variables are constant (public, static, and final) and all methods are abstract and public.

You can use the interface name directly to access static variables declared within it.

Multiple Interfaces

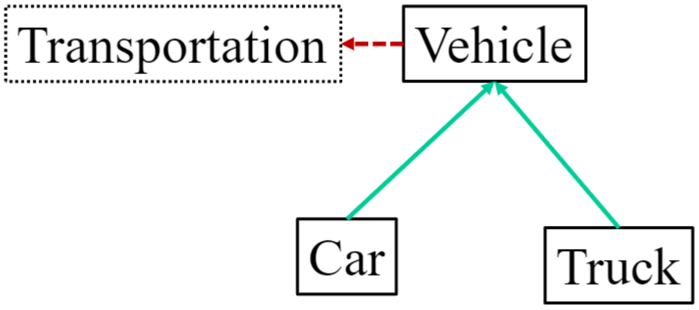
A class can implement multiple interfaces, say you have some interface A and B, then the class C can be declared in the following way to implement them both:

```
public class C implements A,B{
// body of class implementing the methods in the interfaces
}
```

Even if both interfaces have the same method with the same signature (return type and parameters) there is no issue, you just implement one and since there is no body there is no conflict.

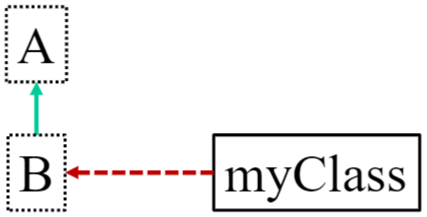
Interfaces and Abstract Classes

If an abstract class implements an interface, then you can leave it to the subclasses to implement the methods in the interface. In this case the subclasses *must* implement the interface methods. The following is a class relation diagram to help understand:



Extending Interfaces

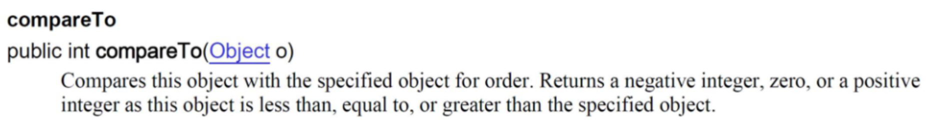
You can make an interface extend another interface. Any class that implements the latter interface must implement all the methods from both interfaces. The following diagram illustrates this:



Built in Interfaces

Many interfaces are built into Java, for example the interface *Comparable* contains one abstract method *compareTo* which compares two objects.

The official description of the method is:



compareTo

public int **compareTo**(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

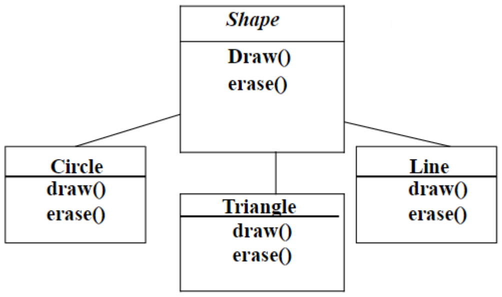
Any class can implement Comparable, to ensure the objects can be compared. The compareTo method returns an integer which is either positive, negative, or zero depending on how the object that called the method compares to the method in the argument. It is up to the coder to implement what that means in the context of each object.

Week 9-11: Polymorphism

Polymorphism is the idea that a method can be called in different ways, depending on what object is calling it. As in the method can be seen in "many forms" (polymorphism).

Java decides which instance of the method to call at runtime, depending on the class of the variable that is calling it. Variable types can be changed at runtime, which is called "late binding". This is done by typecasting.

Take the following example... if you have a class structure with the following methods:



... with the following code:

```
Shape s = new Circle();
s.draw();
s = new Line();
s.draw();
s = new triangle();
s.draw();
```

s is **polymorphic** because it has the ability to take many different forms, and can act like an object of the class it references at the moment. The method call goes to the appropriate class's draw method. If you call a parent class method, then it is the type of the object being reference, not the one doing the referencing, that determines which method is invoked.

Typecasting

An object reference can by type cast into another object reference. This can be to one of its subclasses or superclasses. Any object reference can by type case to type *Object* since *Object* is the superclass to all classes in Java.

There are two types of object casting:

- **Upcast** is when you cast a reference along the class hierarchy from the subclasses towards the root (highest super class). For an upcast, we do not need to explicitly use the type cast operator. This is because by default an object of *Car* is also a *Vehicle*.

- **Downcast** is when you cast a reference along the class hierarchy from the root towards the children or subclasses. Downcasting only works if you are downcasting a generic object type. Otherwise it causes a runtime error. This is because it is **NOT** true that by default a *Vehicle* is also a *Car*.

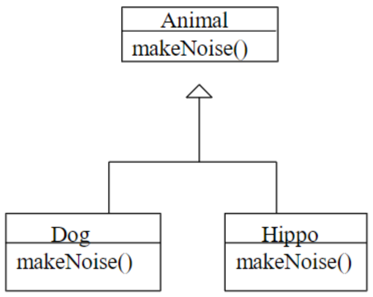
If you do *Vehicle v = new Car()*, with *v* you can only access methods from the car class that are being overridden. Additionally, instance variables do not get overridden, simply they are redefined in the class.

In a method signature, you can also have polymorphic arguments. This is essentially a form of method overloading.

For example in the following code:

```
class Vet {
        public void giveShot(Animal a) {
                a.makeNoise();
        }
}
class PetOwner {
        public void main() {
                Vet v = new Vet() ;
                Dog d = new Dog();
                Hippo h = new Hippo() ;
                . giveShot (d) ;
                v. giveShot (h);
        }
}
```

Assuming the following class structure:



Here, the function call will work even though the input type is not directly animal. This is because Dog and Hippo are subclasses of Animal, and they have overridden the method call.

Late Binding with Interfaces

You can also use an interface as a reference type, but you must instantiate an object of a concrete (non-abstract) class that implements the interface. In this case you can only call methods that exist in the interface definition.

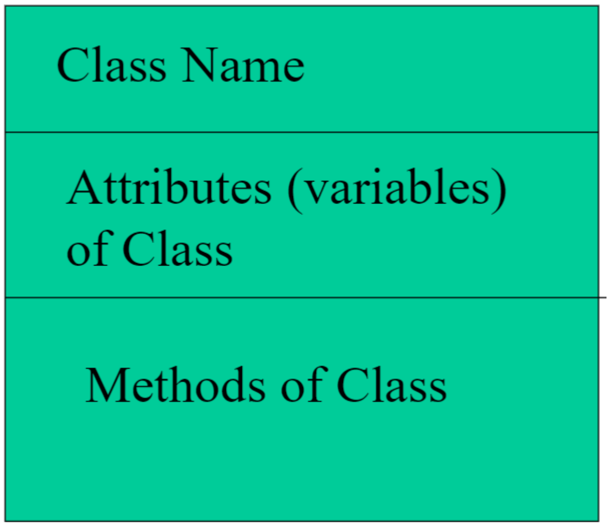
Week 11: Software Design

UML is a standardized general-purpose modelling language used to specify, visualize, construct, and document the design of an object oriented system. An element of this called *Class Diagrams* are what we study in this course.

Class Diagrams

A class diagram described the structure of the systems in terms of classes and objects.

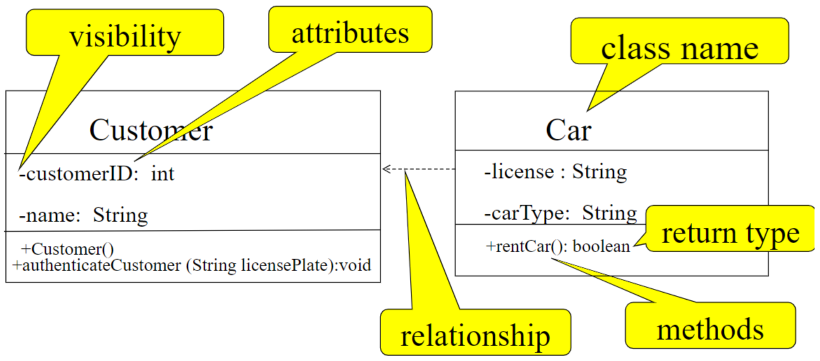
In general the *UML Representation of Classes* is as follows:



Given some classes like the following two:

```
class Customer{
        private int customerID;
        private String name;
        public Customer(){}
        public void authenticateCustomer(String license) {}
}

class Car{
        private String license;
        private String carType;
        public Car(){}
        public boolean rentCar(){}
}
```

We would represent these classes as:



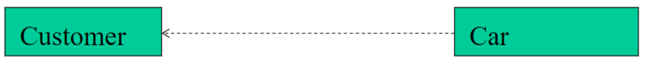
On the diagram you can see that there are visibility modifiers on the method names, the conventions are:

- *Public* + means all objects within system have access.
- *Protected* # means all instances of the implementing class and its subclasses have access.
- *Private* − meaning only instances of the implementing class have access.

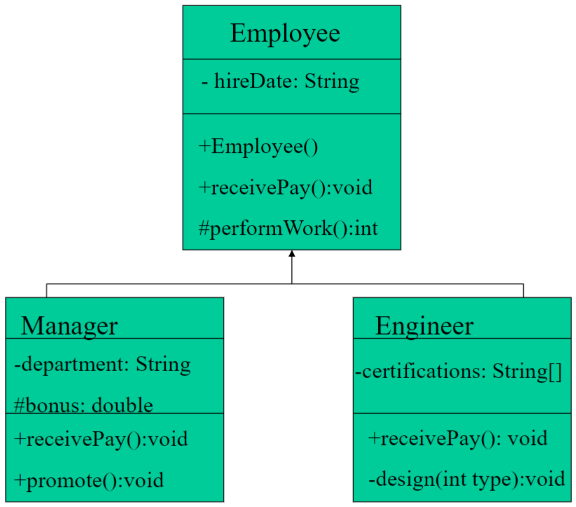
Relationships

There are then 3 types of relationships classes can have:

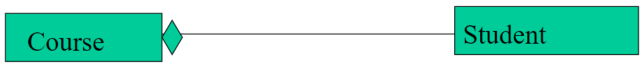
- Dependency relationships.
  - This type of relationship means one class relies on another in some way, usually by calling the methods of the other.
  - This is drawn with a dotted line with an open arrow.



- Generalization relationships.
  - This type of relationship means one class is a subclass of another. This can be thought of as a ”IS-A” relationship.
  - This is drawn with a solid line with a closed arrow.



- Aggregation relationships.
  - This type of relationship means many instances of a class is related to another class. You can think of it like a ”A-PART-OF” relationship.
  - This is drawn with a solid line and placing a diamond nearest the class representing the aggregation.



Additionally, interfaces are denotes with a dotted line and a closed arrow, as we have seen in previous examples.

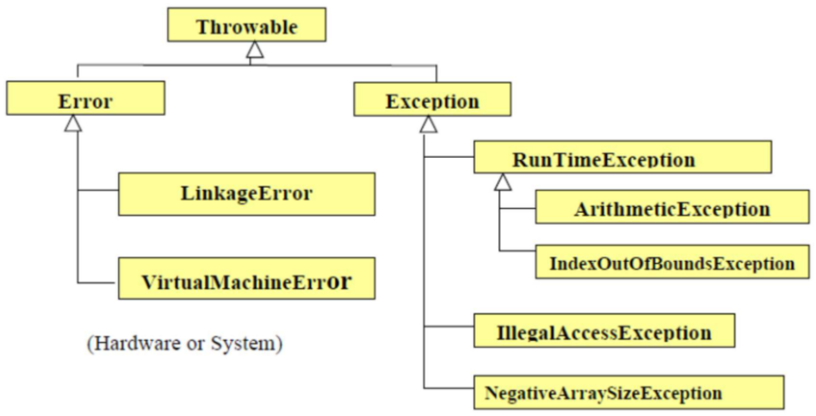
A good way to understand the relationship things have in real life, the following table is a good rule of thumb with examples:

Example	***Grammatical construct***	***UML Component***
“toy"	noun	class
"3 years old"	Adjective	Attribute
“enters"	verb	Operation (Method)
“is a" ,“either..or", “kind of…"	Classifying verb	Inheritance
"Has a ", “consists of"	Possessive Verb	Aggregation

Week 12: Exception Handling

An exception is an event that occurs at runtime which disrupts the flow of the code. For example if the user inputs their name in the age field of a form. Your code should have a way of dealing with these errors without crashing.

The following is the general hierarchy of exceptions:



You can handle an exception where it occurs, or in another part of the program (like another class). Either way we say that code can *throw* and exception, which can then be *caught* by some other code.

Try-Catch Block

A try-catch block is used to try a line of code, and see if it will throw an error, if it does it also describes how the exception should be handled. The following is the syntax:

```
try {
        // Code that might generate exceptions
} catch (Type1 id1) {
        // how to handle code of exception type 1
} catch (Type2 id1) {
        // how to handle code of exception type 2
} catch (Type3 id1) {
        // how to handle code of exception type 3
}
// ... and so on
```

Exception types are in the diagram above, and are self-explanatory. You can also use the generic *Exception* to look for all exceptions. You are declaring an object of these types and so that why it needs a name as well.

The *finally* block is where you can put code that must run no matter if an exception has occurred. This is important for files to close properly. The syntax is:

```
try {
} catch { ...
} finally {
        // code that will always run
}
```

Most of the time you want to throw the exception back at the main block for it to be dealt with there. You do this by the code in the following example:

```
class Age {
        static void throwingDemo() throws IllegalAccessException{
                throw new IllegalAccessException("Just a demo");
        }
        public static void main (String[] args) {
                try{
                        throwingDemo();
                } catch(IllegalAccessException e){
                        System.out.println(e.toString()); }
        }
}
```

Checked Exceptions

Checked exceptions are exceptions which are checked at compile time by Java. They are all subclasses of *Exception* in the diagram above.

Unchecked Exceptions

Unchecked exceptions are exceptions which are not checked at runtime by Java. They are subclasses of *RuntimeException*. These errors include dividing by zero, and index of array out of bounds. These are errors that the program cannot recover from.

Custom Exceptions

You are able to customize the error message which is thrown to the user from the throw statement such as in the following:

```
throw new IllegalAccessException("Just a demo");
```

You are also able to create your own exceptions which can be thrown by using code such as the following:

```
class RideRestrictionException extends Exception{
        public RideRestrictionException(String message){
                super(message);
        }
}
```

When you are trying to catch exceptions in a hierarchy, then you want to catch the most specific types first, and then the super classes. Otherwise the superclass will just catch them all.

Rules in exception handling include:

- You cannot have a catch or a finally block without a try.
- You cannot put code between the try and the catch block.
- A try must be followed by either as catch or a finally.
- A try with only a finally (no catch) must still throw the exception.

Conclusion

This concludes the content covered in this course. I hope these notes were helpful! Good luck on the exam!

- Adam Szava