The haute couture collection has all of our cutting edge and experimental designs.

This file has notes taken from lecture.

For quick and easy references and explanations, try out our autumn-winter 2022 ready-to-wear collection.

Table of contents

    - Objects
- Lect 6: Constructors & Encapsulation
    - Overloaded Constructors & Constructor Chaining
    - Clone/Copy Constructor
    - Memory Diagrams
        - Slide 25
        - Slide 26
    - Garbage Collection Mechanism
    - In-Class Activity
    - Encapsulation
        - Access Modifiers
        - Setters & Getters
        - how do I know what to make private?
        - UML
    - Why OOP?
- Lect 7: Static vs Non-Statics
    - Variables
    - Methods
        - Static Factory Method
            - Why?
- Lect 8: Statics vs Non-Statics cont'd
    - Class Invariants
    - Design by Contract (DBC) and Exception
        - Ex: unchecked exception
    - Try... Catch statement
    - Junits and Exceptions
- Lect 9: Aggregations and Compositions
    - Aliasing
    - Deep Copy
    - Immutability
    - Shallow Copying
- Lect 10: Aggregations and Compositions Cont'd
    - Review
    - Aliasing
    - Deep copy
    - Shallow copy
    - Object Relationships
        - Has-a relationships
            - Aggregation
                - UML
                - Example
            - Composition
                - UML
                - Example
    - Privacy Leak
        - Example

- Lect 11: Inheritance
  - UML
- Lect 12: Inheritance Cont'd
  - Overridden Methods
    - Example
  - How to Recognise an Inheritance Relationship
  - Inheritance and Access Modifiers
  - Designing
  - Non-Extendable Classes
  - Single Inheritance - Deadly Diamond of Death
  - Inheritance Summary
  - Overriding vs Overloading Methods
  - in-class
  - Queues
  - `Object` Class
- Lect 13: Polymorphism
  - Polymorphism
    - avoiding errors
      - Examples
    - Polymorphism Summary
- Lect 14: Static & Dynamic Binding - DBC & Inheritance & Polymorphism
  - Dynamic Binding
  - Static Binding
  - DBC - Inheritance & Polymorphism
    - Recall:
    - Inheritance & Polymorphism
- Lect 15: Exceptions
  - Exceptions
    - What if an Exception happens?
    - How Do We Make Exceptions Help Us?
    - Making An Exception
    - Handling Exceptions
      - `throw-catch`
      - throw without catching
    - User-Defined vs Java Exceptions
    - Exceptions and Inheritance
    - Misc Exception Tips
    - Exceptions and Program Termination
  - `Object` Class Cont'd
    - `toString`
    - `equals()`
- Lect 16: Obligatory Methods (`Object` class)
  - Housekeeping
  - Obligatory Methods
    - `equals()`
      - Overriding

        - Checking Classtype - `getClass()`
        - Applications - Searching Techniques
    - `hashCode()`
        - Hash Tables
        - `hashCode()`
        - Overriding
- Lect 17: Abstraction & Interfaces
    - Abstract Classes & Methods
        - Abstract Classes
        - Abstract Methods
        - Summary
- Lect 18: `Interface`, `Comparable`, and `compareTo()`
    - `Interface`
    - `Comparable`
        - `compareTo()`
- Lect 19: Generics
    - Generic Types
        - Why?
    - Inheritance (1)
        - Array of Generics
    - Inheritance (2)
    - Multiple Parameters
    - Parameter Naming Conventions
    - Generic Methods
        - Wrapper Class
    - Generic Method Cont'd
- Lect 20: Generics Cont'd
    - Bounded Type Parameters
        - Wildcards
            - Upper Bound Wildcards
            - Lower Bound Wildcards
            - How To Use Wildcards

Week 1

Mark Breakdown

	weight	comment
in-class (iCocker)	10%	opens during lecture, usually open till midnight
Labs	10%	10 labs, 1%/each, take home, 1 week

	weight	comment
Programming Exams	40%	2 exams, 20%/each, take home, 1 week
Term test	20%	in-person, written, closed-book
Final exam	^	^

50% rule: Earn at least 50% in written assessments.

Can be cumulative 50% across both written exams.

Your term test mark gets dropped if you're final exam mark is better.

Review

Java Structure

For this section we'll use an example program called `myProgram`.

```
//myProgram.java
public class myProgram{
  variable_type variable_name;

  public static void main(String[] args){
    //code
  } //end of main
} // end of class
```

This concludes java structure basics.

Packages

A package creates a space for a set of classes to exist in together, a name space.

This allows us to have different classes/methods with the same name, we can just put them into a different package.

Packages can contain subpackages that are accessed using the dot, `.`, operator.

- `package_name.sub_package_name`
- `Java.lang`
- `Java.io`

Memory Model

Data Types & Addresses

name	size(bits)	size(bytes)	value range	operators
byte	8	1	(-128)-(127)	+ - * /
short	16	2	(-32,768)-(32,767)	^
int	32	4	(-2^31)-((2^31) -1)	^
long	64	8	(-2^63)-((2^63) -1)	^
float	32	4	floats are approximations and can only be so precise	^
double	64	8	not going into this	^
boolean	1		true or false (0 or 1)	&& (the two lines) !
char	16	2	unicode 0-65535	< <= > >= == !=

There are 8 bits in a byte and each byte takes up an address. If a data type is too big to fit in 1 address then it takes up however many addresses it needs. If a data type is too small to fill out a whole address it still occupies that whole address.

When a variable of a certain data type is defined, the correct amount of spaces are reserved to accomodate the data.

Examples

Primitives

Code:

```
int height = 165;
double weight = 58.5;
char initial = 'D';
boolean found = true;
```

var name	memory address	memory spaces	size(bytes)
height	100	165	4
weight	104	58	8
initial	112	'D'	2
found	114	true	1

Non-Primitives/References: Arrays

With non-primitives/references we're actually pointing to another object in the memory elsewhere in the garbage collectible heap.

```
int [] height = {165, 170};
double [] weight = {58.5, 70.4};
char [] initial = {'D', 'A'};
boolean [] found = {true, false};
```

var name	memory address	memory spaces	memory area
height	100	200a	Stack
weight	*	*208a	
...	...	...	...
	200		GCH

*In the case of non-primitive reference types like arrays there's no universal amount of reserved space for storing the addresses. In lieu of that certainty we just use notation to denote that these are separate objects in the memory, which objects belong to who, and that everything that is a reference type is in the garbage collectible heap.

For the variables `height[]` and `weight[]` we store the address where the actual data is stored as opposed to the actual data. This is mainly because the size of the actual data may change and having that in the main stack is a headache. The address that we store points to somewhere within the GCH.

Function Calls

Whenever a function is called in Java, a new memory space is created on the stack that we refer to as the function stack. Any variables created inside of there, no matter what we do with them, are deleted from the function stack once the function call is over. They get deleted along with the rest of that memory stack.

Most of the time we pass an argument into function or method call. Different things happen (pass by value vs reference) from a memory perspective depending on the data type (primitive vs non-primitive).

Pass by Value

When we pass in a primitive like an `int` or `double` as an argument to a function Java will copy that primitive into a memory address inside of the function stack. That object exists in the main stack as well as the function stacks.

Pass by Reference

When we pass in a non-primitive like an `array` as an argument to a function Java will copy the reference stored in the main stack into a memory address inside of the function stack. That object exists in the GCH however there are two variables that point to its address during the function call, the variable on the main stack and the variable inside of the function stack.

Documentation - JavaDoc

here are some non-documentation pro-tips:

- give your variables meaningful names that explain their use
  - `student_age`
- follow conventions
  - use for loops to scan an array instead of a recursive function
- write comments for when code becomes complex
- you can mark where a large piece of code ends and another begins

Documentation is when we write shit down to make sure we know what the fuck is going on.

In Java we make use of JavaDoc.

To use JavaDoc for a method, function, or class you create a block comment on the line above your method with the first line being `/**`

Here are the most common/useful bits:

- `@author` and `@version` state the obvious
- `@param` describes inputs
- `@return` describes the output

- JavaDoc makes use of some html style syntax
  - `<p> </p>` makes a paragraph
  - `<code> </code>` will put a different typeface on export to show that you are referring to a piece of code

Many java focused IDEs, as well as general use IDEs with the help of plugins, have an option to generate JavaDoc block comments for your code as well as export an html api.

Testing & JUnit

Testing is when we run shit around to make sure we know it actually fucking works.

There are 3 types of errors

- compiler errors
  - syntax errors
- run time errors
  - dividing by zero
- logical errors
  - skill issue

Since the dawn of programming we have used the time tested technique of calling a function in main and seeing what we can throw at it. This requires a human to write tests and verify results making for a time intensive, error-prone mess of a process.

Instead we automate using JUnit tests.

Unit testing is the practice of testing the smallest functional units of your code (individual methods and functions) using tests that run automatically and automatically return human readable outputs.

JUnit is a unit testing framework for Java. To use it you have to define a class (just like everything in Java), import the libraries relevant to your testing, and write a method under the test that is called.

Most Java focused IDEs have an option to generate a test file automatically.

Unit testing still requires a human to write tests.

Design by Contract

DBC is the idea that there is a contract between the coder/implementer(you) and the user/client.

The contract states:

- The code will only work if:
    1. The client passes the methods expected by the implementer (pre-condition)
    2. the output is what's expected by the client (post-condition)

You write the contract beforehand and you don't have to say shit when they try to pass `"two"` into `fast_bitsqrt()` when the pre-condition said to only pass in doubles.

I deleted the lecture 1 and 2 lecture notes so the above are a recreation.

Lect 3: intro to recursion

Function Calls and Stack Frames

When a function is called, a new stack frame is created for the local variables to be stored. When the function is returned or otherwise finishes, the stack frame is deleted and the memory is freed up again for other programs to use.

Once a memory space is gone, it's gone forever. People die when they are killed

Recursion

Recursion is when a function calls itself. With each call we have a new stack frame. Stack overflow occurs when there is too many frames for the memory to handle (infinite recursion). Recursion requires that each case eventually move towards the base case so that we stop creating stacks. Once we stop creating stacks we can go back and start popping off the stacks we created beforehand.

Some problems are recursive in nature (repeating a process with slightly different inputs over and over again, those inputs being the outputs of previous iterations and the ultimate answer resulting from going back and using all the results together). Therefore it is easier and more understandable to present with them with recursion.

It's a new mode of thinking.

Factorials, finding greatest commmon divisor, search and sort algos, folders containing other folders, fractals, and many other things are examples of things that are just recursive in nature.

Tracing a Recursive Algo w/ Examples

Ex: Is the input a palindrome?

Plaindrome - a word that is spelt the same left to right as it is right to left

Empty and single character strings are palindromes(albeit crappy palindromes).

"racecar" is still "racecar" backwards.

"racer" isn't.

```
public static boolean isPalindrome(String input) {
    boolean palindrome = false;
    if (input.length() <= 1 )
        palindrome = true;
    else
        palindrome = (input.charAt(0) == input.charAt(input.length()-1)) &&
isPalindrome(input.substring(1, input.length()-1));
    return palindrome;
```

Basics of Recursion

In a recursive function there are 2 kinds of cases. A base case and a recursive case.

A recursive function must have at least one of each.

If there is no recursive case then it's just an if-else gate.

If there is no base case then we'll keep recursing until we run out of memory.

Every recursive case has to move towards the base case at some point or at least move towards a different recursive case that will meet the base case at some point.

How to Design a Recursive Algo

Divide & Conquer (top-down approach):

- 1 & n-1 (buttom up)
  - solve the problem for n input, using the sol'n for the n-1 input
- find out how to solve the smallest/easiest case w/o recursion
  - summing up an array of n-elements?
  - summing up an array of 1 element is easy
  - make recursive cases making the arrays smaller and smaller until we hit the base case

Lect 4: Recursion Cont'd

Divide and Conquer - Find the sum of all the entries of an array

If the array is 1 element then the sum of all the elements is that element

If the array is 2 elements then the sum of all the elements is the first element plus the second element.

We could also divide the array in 2 in this case then add the first element of each array to one another.

If the array is n elements long, divide the n elemnt long array into 2 subarrays, add their first terms, if there are any remaining whose sum cannot be taken by our first rule then divide those arrays in 2 once more adding their first terms.

we can divide the problem space in 2 in order to make it smaller and smaller (divide and conquer)

Recursion is not always the best sol'n

Recursing requires the reservation of a lot of memory and the cost of memory as we go deeper into the recursion scales steeply.

Especially when compared to other more straightforward sol'ns that don't require recursion and/or as much memory reservation.

Recursion is also a pain in the ass to grapple with and design.

Misc recursion notes

Recursion is a different mode of thinking so it's hard to come up with a sol'n right away.

It is possible to have more than one base and/or recursive case

Problems solved via recursion can be solved iteratively as well

- for loops
- while loops

Sometimes you neeed a helper function for recursion.

Even if a problem is recursive in nature, you don't necessarily need to resort to recursion.

Closing

Every recursion must have a base case and a recursive case

Use divide and conquer and bottom up methods

Linear recursion calls itself once in every iteration. Binary recursion calls itself twice in every interation.

Lecture 5: Finally Advanced Object-Oriented Programming

Adv oop is more about concepts. We already know all the base concepts and required syntax.

Variable Types

Java cares about the type. **Java is a strongly typed language.**

WHen you create a variable in Java, memory has to be reserved in order to store that variable. If you want an integer then java sets aside 4 bytes, if you want a double then java sets aside 8 bytes, and so on.

Some data types are flexible in size like arrays. At compile time java doesn't know the total required memory space for these data types. At runtime it fucks around and finds out.

Examples:

- Strings
  - `userName` could be any number of charcters long

THese are nonprimitives

Non-primitives

Data types that have flexible memory sizes that aren't known at compile time.

Examples:

- String
- Math
- Scanner
- ArrayList
- Vector
- StringBuilder
- StringBuffer

`ArrayList` and `Vector` have similar functionality but are very different. Same with `StringBuilder` and `StringBuffer`.

Non-primitive types are also called `reference types`. They don't actually hold the data related to them, they hold the memory address for where the heap of the actual data is.

```
public static void main (String[] args) {
    double grade = 98.4;
    String name = "Marzieh";
}
```

the `double grade` stores the `98.5` in the fixed memory aka program stack. `grade` actually holds the value.

The `String name` stores the `"Marzieh"` in the growable memory spaces aka garbage collectible heap. `name` holds the address where the data is stored.

Reference types have zero or more states (represented by their `instance variables` aka `fields` or `attributes`). These associated instant variables make up the "state" of the reference type variable.

There are zero or more behaviours associated with the non-primitives.

Classes

A `class` is a bundle of instance variables and methods. Bundling together functions and information.

It's also a non-primitive data-type and so has the same traits as other non-primitives. Flexible memory allocation, unknown required memory allocation at compile time, zero or more states, and zero or more behaviours. When we write a `class` in java we're creating a blueprint for creating an `instance` of that class, this is called an `object`. You can't use the class on it's own, you have to make an `instance`.

Presenting a Class

To present a class we use notation called **Unified Modelling Language (UML)**

Class Name
Instance variables
methods

selfDrivingCar
Make: String
Model: String
color: int
plateNumber: char[]
start(): boolean
accelerate(int): void
changeGear(int): void
brake(): boolean
turn(char): void

When the designer (us) makes this, they can then give it to the programmer (also us) to be created easily.

Creation of a Class

Recall: `public` classes are accessible by everyone and everything.

Access modifiers and static vs non-static is covered later

```
public class className {

  // declare instance and class variables

  // make a constructor method, there can be more than one
  // <access modifier> <className>(<arg1 type> <arg1 name>, etc.)
```

```
  public className(){
    // constructor code
  }

  // instantiate methods
  // <access modifier> <static> <returnType/void> <method name>(<arg1 type> <arg1
name>, …)
  pubic static int getNum(){
    //method code
  } // end of method

}// end of class
```

Objects

to make an `object` (`instance` of a class):

```
//<class name> <variable name> = new <class name>();
person jane = new person();
```

Accessing the `object`'s property (`field`s and `method`s) uses the `dot` "." operator.

```
System.out.println(jane.name)
```

`jane` is stored in the stack but what she stores is a memory address. That memory address points to the `GCH` (garbage collectible heap). At the address all the data is stored.

read about constructors

Lect 6: Constructors & Encapsulation

Constructors are special methods:

- they return no value
  - they are not void
- it is the same name as the class
- it is used to initialize the instance variables

We can initialize instance variables by making a constructor that accepts several arguments. In doing so we can avoid rewriting similar lines of code over and over again.

When writing class methods (even and especially the constructor) `this` allows us to refer to the current object. `this` followed by the `dot .` operator will allow us to take variables and methods from the current object specifically.

When making an instance of an object whose code makes use of `this.` then it gets translated in a way. An instance of the class `student` that we call `jade` will turn statements like `this.name` into `jade.name`. *This refers to the `object` not the `class`.*

Overloaded Constructors & Constructor Chaining

A class can have multiple constructors in order to give flexibility in creating an object. Sometimes you don't have to specify a certain aspect of an object or don't want to, in these cases we can have constructors that take more or less or different arguments. When we use those specific constructors then we can easily differentiate it in our use cases. This is called *overloading constructors*.

If we have duplicate code b/w all of our constructors then we can call other constructors inside of our constructor. A basic constructor then other more advanced/specialized constructors depending on the args. We can call a constructor inside a `class`'s code by using `this(arg1, arg2, arg3, ..., argN)` where the list of args are the required args for the constructor we want. This is called *constructor chaining*.

Clone/Copy Constructor

A `class` constructor that takes an existing `object` instance of the `class` as the input argument then copies over the attributes to a new instance of the class, creating a new `object` in the process.

Memory Diagrams

Slide 25

Class code:

```
public Student (Student st) {
this.initial = st.initial;
this.studentId = st.studentId;
this.enrolmentYear = st.enrolmentYear;
}
```

Our code using the `Student` class:

```
Student alice = new Student('A', 1256, 2016);
Student rose = new Student(alice);
System.out.println(alice.initial);
System.out.println(rose.initial);
alice.initial = '#';
System.out.println(alice.initial);
System.out.println(rose.initial);
```

memory diagram:

address	stored	
100	2000a	alice
	3000a	rose
...	...	...
2000	A	initial
	1256	studnetId
	2016	enrolmentYear
...	...	...
3000	R	intial
	1256	studentId
	2016	enrolmentYear

Slide 26

```
// rose is an object of Student
rose.initial = 'R';
rose.studentId = 1000;
rose.enrolmentYear = 2020;
Student julia = rose;
System.out.println(julia.initial);
System.out.println(rose.initial);
rose.initial = '#';
```

```
System.out.println(julia.initial);
System.out.println(rose.initial);
```

memory diagram:

address	stored	
100	2000a	rose
	2000a	julia
...	...	...
2000	A	initial
	1256	studnetId
	2016	enrolmentYear

We didn't use the keyword `new` and just directly assigned an `object` to another `object`. New memory space was not reserved so they just point to the same point in memory.

Garbage Collection Mechanism

If we have something stored in the GCH but isn't referenced by any variable then it's no longer needed. Java will get rid of this for us so that we can free up memory. C and C++ don't do this.

Active references point to a memory address, null references don't point to anything. Reachable objects give access to their components in memory, non reachable can't be reached by their objects, they aren't being pointed to so they can be garbage collected.

Garbage collector finds these unreachables and declares them as free space

Let's take a look at this code:

```
Student john = new Student();
Student jane = new Student();
john = jane;
jane = null;
```

Line 1:

address	stored	

address	stored	
100	2000a	object `john`
...	...	...
2000	<all of `john`'s stuff>	

Line 2:

address	stored	
100	2000a	object `john`
	3000a	object `jane`
...	...	...
2000	<all of `john`'s stuff>	
...	...	...
3000	<all of `jane`'s stuff>	

Line3:

address	stored	
100	3000a	object `john`
	3000a	object `jane`
...	...	...
2000	<all of `john`'s stuff>	
...	...	...
3000	<all of `jane`'s stuff>	

Line4:

address	stored	
100	3000a	object `john`

address	stored	
	null	object `jane`
...	...	...
2000	<all of `john`'s stuff>	
...	...	...
3000	<all of `jane`'s stuff>	

we find that the data stored in `2000` which was `john`'s old data is unreachable so it's garbage collectable.

In-Class Activity

```
Car herCar = new Car();
Car hisCar = new Car();

Car myCar = hisCar;
hisCar = null;
myCar = herCar;
herCar = hisCar;
```

address	stored	
100	null	herCar
	null	hisCar
	2000a	myCar
...	...	...
2000a	< `herCar` stuff>	
...	...	...
3000a	< `hisCar` stuff>	

Encapsulation

Data encapsulation refers to hiding the internal states of an object in oop. An `implementer` controls the `clients` access to the instance variables.

We have to make these instance variables inaccessible to the clients.

THis can be done by choosing the correct access modifiers.

Access Modifiers

Access Modifier	class	Subclass Same Package	Subclass Outside Package	package	World (outside the package)
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	N	N
no access modifier	Y	Y	N	Y	N
private	Y	N	N	N	N

`public` means anyone can access it

`protected` means that we can't access it in other classes whether they're in the same package or other packages. Only accessible by subclasses

`no access modifier` just means you write nothing. Accessibly by any other class in package as well as subclasses in the same package.

`private` makes it inaccessible outside of the class.

Setters & Getters

`get&lt;Var> &lt;set>Var`

When data is encapsulated, the only way you can get access is via the object methods.

Accessors (getters): return the value of the instance variable.

Mutators (setters): updates the value of the instance variable.

how do I know what to make private?

I don't konw.

It's up to the programmer/designer/contractor/boss's discretion.

If you are not sure how sensitive a data is, make the variable invisible and provide the access by mutator and accessor methods.

It's easier to change things that way.

UML

In UML public members are shown by `+` while private are shown by `-`

Why OOP?

Bundling code into individual softare objects provides a number of benefits, including:

- modularity
- information-hiding
- code re-use
- pluggability and debugging ease

Lect 7: Static vs Non-Statics

Was late so I have to go look at the video.

Variables

When we place variable declaration outside of any methods or constructors in a class, they become instance variables when we make an instance of the class. Suppose we wanted a counter of how many instances of that class we've created, in this case we use the `static` keyword in declaration. Otherwise a copy of the variable gets copied to the new instance's memory space.

Using `static` makes the variable belong to the `class`, a class variable, as opposed to an individual `object`.

These are normally used as a counter of the objects created but there are other use cases.

The static variable is only initialized when the class is first loaded.

If you do not initialize the static variable, it will get the default value:

- Primitive integers → 0

- Primitive floating point → 0.0
- boolean → false
- Object reference → null

Only make static variables with good reason.

These variables can be accessed using the `dot`, `.`, operator on the `class` or any given `object` of the class.

```
//objectName is an object of className
System.out.println(className.staticVar);
System.out.println(objectName.staticVar);
```

object	var	val

Methods

`static` methods function the same on all instances.

`Math.sqrt(a)`, `Math.pow(a,b)`, and `Math.min(a,b)` all function the same no matter what kind of `Math` class you create. They only function based off of the arguments and class variables rather than the instance variables.

`static` methods like `static` variables can be called with the `dot`, `.`, operator next to the class name or an object.

```
className.methodName(arg1, arg2);
objectName.methodName(arg1, arg2);
//will return the same result
```

Static methods cannot use non-static instance variables or call non-static methods.

Static Factory Method

A method that creates a new `object` of the class in which the static method is defined and returns it.

Normally the name is `getInstance()`.

Why?

This is useful for instances where we have 2 constructors who only differ in a single variable but those variables don't differ in type, making it impossible for the compiler to figure out which constructor is being called.

In this case we can use different versions of `getInstance()` and a private constructor method.

Suppose we want a class that can only have one instance. We could make the constructor private and only have a `getInstance()` that throws an exception if it already exists.

Lect 8: Statics vs Non-Statics cont'd

This became online all of a sudden for some reason. I didn't pay attention. Video starts at slide 23/39.

`toString()` returns the address of any object that it's called for.

Class Invariants

Recall *Design by Contract*:

- clients (users) guarantee the preconditions
- implementers (coders) guarantee the postconditions
- we design with this contract in mind and assume that it will always be fulfilled

Invariants is another component of design by contract.

An invariant is a statement that is true during a certain point in a program's execution.

Generally, an invariant is a certain property that is assumed to be true on the entry of a method and guaranteed to be true on the exit of the method.

Invariants are used for program correctness.

example:

```
/**
 * This method creates a string by inserting the given string in the current arraylist at the given index.
 * <p> please not that @pre and @inv are not the standard javadoc tags. @pre shows that precondition and @inv represent the invaraiant.
 * @param str is the string that is inserted in the current string.
 * @param position is the index in which the given string is inserted.
 * @return retuns  a string that contains the given input string.
 * @exception IllegalArgumentException
 * @pre position >=0
 * @inv <code> string </code> should remain unchanged (immutable). Meaning that the value of the instance variable at
 * the exit of the method, should remain the same as its value at the entry to the method.
 */
public String insertInBetween(String str, int position) {
    if (position < 0) throw new IllegalArgumentException();
    String result = "";
    for (int i = 0; i<position; i++)
        result = result + string.get(i);
    result = result + str;
    for (int i = position; i<this.string.size(); i++)
        result = result + string.get(i);

    return result;
}
```

Here we see that the invariant is that the string should remain unchanged. If our invariant is changed then we know the program is incorrect.

Design by Contract (DBC) and Exception

If the pre-condition isn't met by the client then we want to do something about it. We design by contract and the contract has been violated.

We can throw an exception to stop the program from executing and generate a message about the problem that occured.

3 types of error:

- compiler/syntax error
- logical error
- runtime error

We use these exceptions to catch runtime errors.

Ex: unchecked exception

```
/**
 * This method computes the area of a circle, whose radius is geven
 * @param radius is the radiusof a circle
 * @return the area of the circle
 * @pre radius shold be a number
 * @pre radius should be positive. Zero is accepted
 * @exception throws NumberFormatException
 */
public double getArea (double radius) {
```

```
    if (((Double) radius).isNaN()) throw new NumberFormatException();
    return radius*radius*Math.PI;
```

Entering a non-number or a negative number violates the contract so we throw an exception.

Try... Catch statement

Previously we identified the exception and just threw it to shutdown the program.

We can also decide to handle the exception by using a try-catch statement.

```
try {
    // the rsiky code that may produce a run time error goes here.
} catch (Exception e) {
    // ma proper error message or any other statement that should be run in case exception happens goes here
}
```

```
/**
 * This method squares the value of the `array` at the given `index`
 * @param array a double array
 * @param index an index of the array
 * @exception ArrayIndexOutofBoundsException is thrown in case the index is not in the range of zero to array.length()-1
 */
public void squareIt(double[] array, int index) {
    try {
        array[index] = Math.pow(array[index], 2);
    } catch (ArrayIndexOutofBoundsException e) {
        System.out.println("Invalid index!");
    }
}
```

Junits and Exceptions

Recall:

- there should be a test case from every categary of epected input
- there should be boundary/edge cases

Now we need to also add at least one test case to check the exception.

Normal test cases look like so:

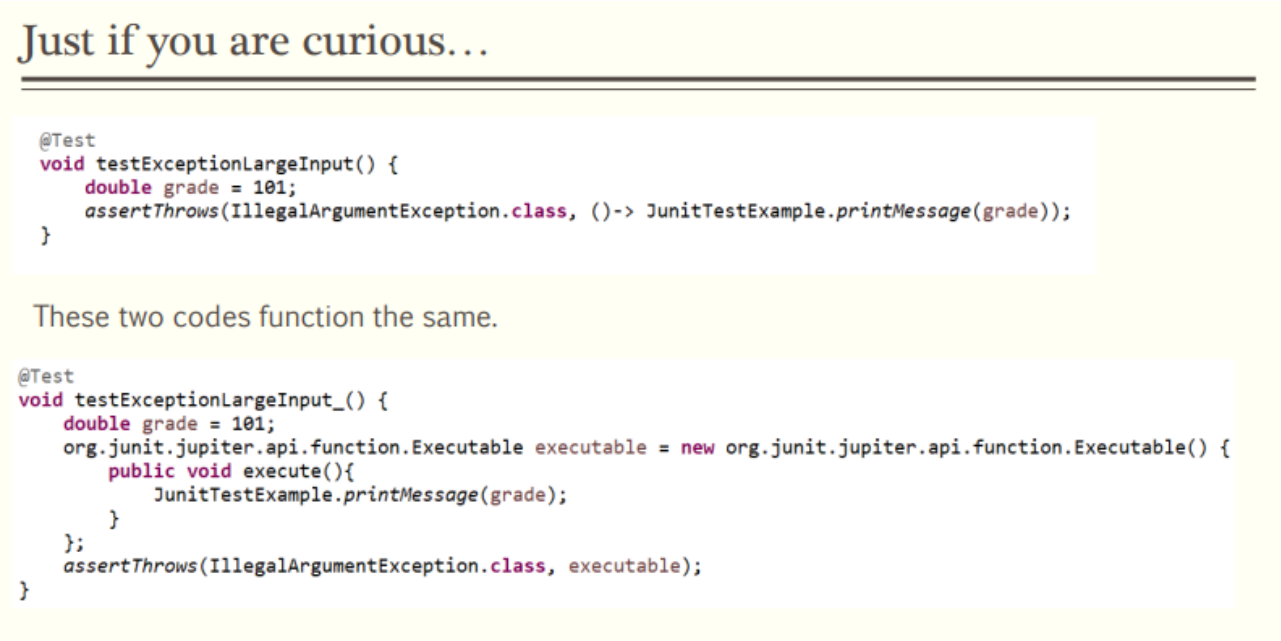
```
@Test
void test1() {
    String expected = "rightAnswer";
    String actual = testedClass.testedMethod(arg);
    assertEquals(expected, actual);    //we assert that expected is equal to actual
or else the test fails
}
```

Testing for exceptions looks like this:

```
@Test
void testException() {
    assertThrows(<name of exception>.class, ()-> testedClass.testedMethod(arg));
}
```

There are a lot of different kinds of exceptions that you will have to throw or attempt to handle.

`()->` is lambda syntax and allows us to not do a fuck ton more typing than we need to.



Lect 9: Aggregations and Compositions

With `reference` data types there is aliasing and deep copying that you need to contend with when it comes to making copies.

This doesn't exist for `primitive`s (`int`, `double`, etc) since they store the data itself at their memory address as opposed to a reference to a memory address.

Aliasing

Aliasing is the creation of a second copy of a `reference` variable using `assignment` operator.

They are both `reference`s but point to the same data in the memory.

```
int[] arr = {1, 2, 3, 4};
int[] arrClone = name;
```

Memory diagram:

mem. address	data	what is being stored
0	100a	Address of data for `arr`
	100a	Address of data for `arrClone`
...	...	...
100a	1	data for `arr`
	2	
	3	
	4	

Deep Copy

Deep copying is making a new `reference`, reserving a new space in the memory, then filling that space with identical data to the data in the memory space referenced by the reference being copied.

```
String arr = "John";
String arrClone = new int[arr.length];
for (int i = 0; i < arr.length; i++)
    arrClone[i] = arr[i];
```

Memory diagram:

mem. address	data	what is being stored
0	100a	Address of data for `arr`
	200a	Address of data for `arrClone`
...	...	...
100a	1	data for `arr`
	2	
	3	
	4	
...	...	...
200a	1	data for `arrClone` (copy of `arr` data)
	2	
	3	
	4	

Immutability

If the state of an object cannot change after it is constructed, it is said that the object is immutable.

You can create an immutable object if you do not let any method change the state of the object.

- no mutator methods
- deep copy the reference variables in the accessor methods
  - no accessor method should return a reference to an instance variable

```
public char [] getPlatNumber() {
    char [] plateNumberCopy = new char[this.plateNumber.length];
    for (int i = 0; i < this.plateNumber.length; i++)
        plateNumberCopy[i] = this.plateNumber[i];
    return plateNumberCopy;
```

Shallow Copying

Shallow copy clones an `object` and it's components but not it's sub-components.

Lect 10: Aggregations and Compositions Cont'd

Review

Suppose we have an object `ob1` with `data1` that points to `a100`

Aliasing

`ob2` is an alias of `ob1`. It points to `a100` which has `data1`

Deep copy

`ob2` is a deep copy of `ob1`. It points to `a200` which has `data2`. `data1` and `data2` are identical.

SUppose that `data1` is something like an arraylist that further references somewhere else `r100`.

Shallow copy

`ob2` is a shallow copy of `ob1`. It points to `a200` which has `data2`. `data1` and `data2` point to `r100`.

Object Relationships

Objects have relationships with one another in oop. This makes code reuse possible.

Types of relationships:

- Aggregation: Has-a relationship
- Composition: Has-a relationship

- Inheritence: Is-a relationship

Has-a relationships

Has-a relationship - When an object of type `A` uses the properties (instance variables and methods) of an object of type `B`

We're describing a situation where an object has an *instance variable* whose type is non-primitive.

There are two types of these relationships:

- aggregation: weak association - the components can exist independent of the object
- composition: strong association - the components cannot exist independent of the object as they are owned by the owner object

Difference in programming (providing access to the components of the class):

- Aggregation uses aliases or shallow copy to create the object
- composition: uses deep copy to crate the object

Aggregation

The relationship between objects is weak:

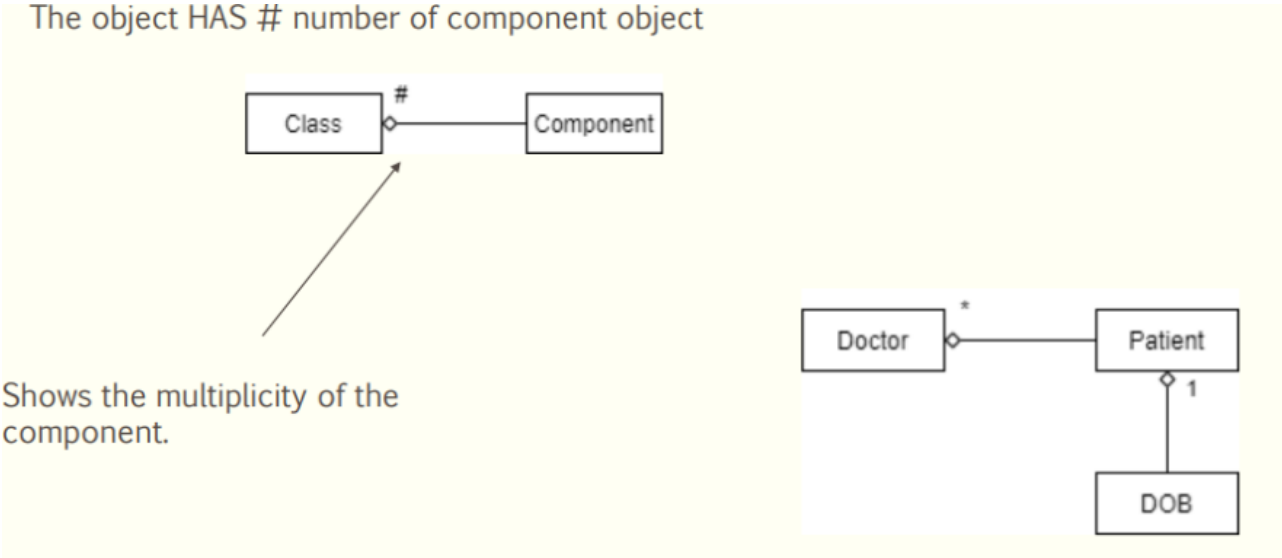
- a student has associated courses
    - what happens when they graduate?
        - the course still exist
- a doctor has patients
    - what happens when they retire?
        - the patients still exist

Aggregation means that the object does not own its component.

The component exists independently of the object.

Once the object is gone(cleaned up from the memory), the component is still there.

UML



draw a line from the component to the class ending with a hollow diamond and marked with the number of that class belonging to the other class or an * if the number can be more than one.

Example

```
public class Patient {
    String name;
    Calendar dob;

    public Patient(String name, Calendar dob) {
        this.name = name;
        this.dob = dob;
    }
}

public class Doctor {
    private String name;
    private ArrayList<Patient> patient;

    public Doctor(String name, ArrayList<Patient> patient) {
        this.name = name;
        this.patient = new ArrayList<Patient>();

    }

    // Aliasing/shallow copy
    // we don't want to make a new patient, this patient just exists
    public Doctor (Doctor doctor) {
        this.name = doctor.name;
        this.patient = doctor.patient;
    }
```

```
    //Mutators
    public void setName(String name) {
        this.name = name;
    }

    public void setPatient(ArrayList<Patient> patient) {
        this.patient = new ArrayList<Patient>();
        for(int i = 0; i < patient.size(); i++)
            this.patient.add(patient.get(i));
    }

    //Accessors
    public String getName() {
        return this.name;
    }

    Public ArrayList<Patient> getPatient() {
        return this.patient;
    }
}
```

address	objects	what is stored
0	doctor	100a
	patient1	200a
	patient2	300a
…	…	…
N	patientN	N00a
…	…	…
100	patient	1000a (ArrayList)
…	…	…
1000	patient1(alias)	200a
	patient2(alias)	300a
…	…	…
100N	patientN(alias)	N00a

for mutators (setters) we use shallow copy

For getters (accessors) we use aliasing

Composition

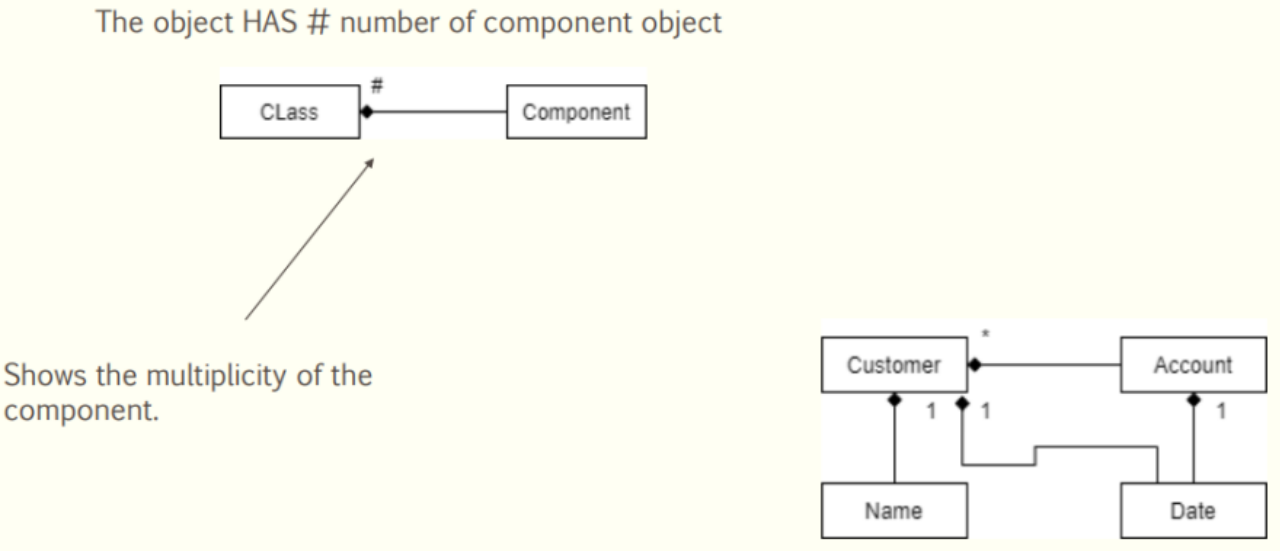
The relationship between objects is strong:

- the object owns its components
- the object has exclusive access to its component
- the components cannot live independent of the object
  - fun fact: "kill" is a technical word in the world of programming and computers
    - we kill a process
    - we kill something in the memory
    - we kill an object
- if the object is cleaned up from the memory
  - all of the components will be gone too

examples:

- a `house` has-a couple of `room`s
  - do the `room`s exist when the `house` is demolished?
- a `client` at a bank has-a couple of `account`s
  - the `account`s are gone when the `customer` leaves

UML



draw a line from the component to the class ending with a solid diamond and marked with the number of that class belonging to the other class or an `*` if the number can be more than one.

Example

```
public class Account {
    char accoutnType;
    int accountNumber;
    double balance;
    Date dateOpened;
}

public class Customer {
    String name;
    Calendar dob;
    ArrayList<Account> account;

    public Customer(String custName, Calendar dofb, ArrayList<Account> acc) {
        this.name = new String (custName);
        this.dob = Calendar.getInstance();
        this.dob.set(dofb.get(Calendary.YEAR), dofb.get(Calendar.MONTH),
dofb.get(Calendar.DAY_OF_MONTH));
        this.account = new ArrayList<Account>();
        for (int i = 0; i < acc.size(); i++)
            this.account.add(new Account(acc.get(i)));
    }

    //Mutators
    public void setName(String name) {
        this.name = name; //String is an immutable so we didn't have to make a new
object
    }
    public void setDob(Calendary dofb) {
        this.dob = Calendar.getInstance(); //make a new calendar object
        this.dob.set(dofb.get(Calendar.YEAR), dofbget(Calendar.MONTH),
dofb.get(Calendar.DAY_OF_MONTH));
    }
    public void setAccount(ArrayList<Account> acc) {
        this.account = new ArrayList<Account>();
        for (int i = 0; i < acc.size(); i++)
            this.account.add(new Account(acc.get(i)));
    }

    //Accessors
    public String getName() {
        return this.name;
    }
    public void getDob() {
        Calendar db = Calendar.getInstance();
        db.set(this.dob.get(Calendar.YEAR), this.dobget(Calendar.MONTH),
this.dob.get(Calendar.DAY_OF_MONTH));
```

```
    }
    public void setAccount() {
        ArrayList<account> acc = new ArrayList<Account>();
        for (int i = 0; i < this.account.size(); i++)
            acc.add(new Account(this.account.get(i)));
        return acc;
    }
}
```

address	objects	what is stored
0	customer	100a
...	...	...
100	account	1000a (arrayList)
...	...	...
1000	acc1	10000a (all the properties)
	acc2	20000a (all the properties)
	acc3	30000a (all the properties)
...	...	...
N000	accN	N0000a (all the properties)

if `customer` gets cleaned up from the memory then it cleans up everything that it's pointing too as well. So the `account` array list variable is gone and all of the accounts inside of it (`acc1`, `acc2`, `acc3`,... `accN`).

In `setName()` we didn't have to create a new string object because even though `String` is non-primitive it is immutable.

If a non-primitive is immutable (if we made `ArrayList`, `Account`, and `Calendar` immutable) then we could do the same thing as well.

Privacy Leak

a situation where a client get access to the data that they should not get access to it.

Privacy leak happens when a class exposes a reference to an attribute, which was not supposed to be public.

This only applies to non-primitive attributes. Primitive and immutable attributes are privacy leak resistance.

Example

We have the existing patient objects in the memory. We make a private list of patients for the doctor class.

If we implement the relationship b/w the doctor and patients as an aggregation (doctor owns an alias of the patients) then we get a privacy leak.

We now know all of the new things happening with the patients just by looking at the patient objects that are existing in the memory already.

We have to implement as a composition so that we can protect the information regarding the new things happening with the patients.

Lect 11: Inheritance

This is a continuation of the last week's lecture with object relationships

Inheritance is the other type of object relationship, an Is-A relationship.

Different kinds of objects often have a certain amount in common with each other. Shapes all have a perimeter and an area. Medical practitioners all have names, registration numbers, and the ability to get hte history on a patient.

With inheritance we can create a piece of code that objects have in common then reuse that code for a ton of other objects. We place this common code in a class called `superclass` or `base class`. The specific states and behaviors of an object stays in their own class, which is now called a `subclass` or `derived class`.

When two(or more objects) share some attributes and methods, an IS-A relationship is created:

- `subclass` IS-A type of `superclass`

Subclasses inherit all the public and protected attributes and methods of their superclass.

UML

ALl the attributes an dmethods are written in UML as before.

`+` and `-` are used for public and private access modifiers.

Protected features are shown with #.

- protected features are visible to and usuable by:
  - the same package subclass
  - same package non-subclasses
  - different package subclasses
- but not by a different package non-subclass

Lect 12: Inheritance Cont'd

Overridden Methods

Subclasses can make their own methods that have the same name as the superclass' methods but have different functionality to better suit it.

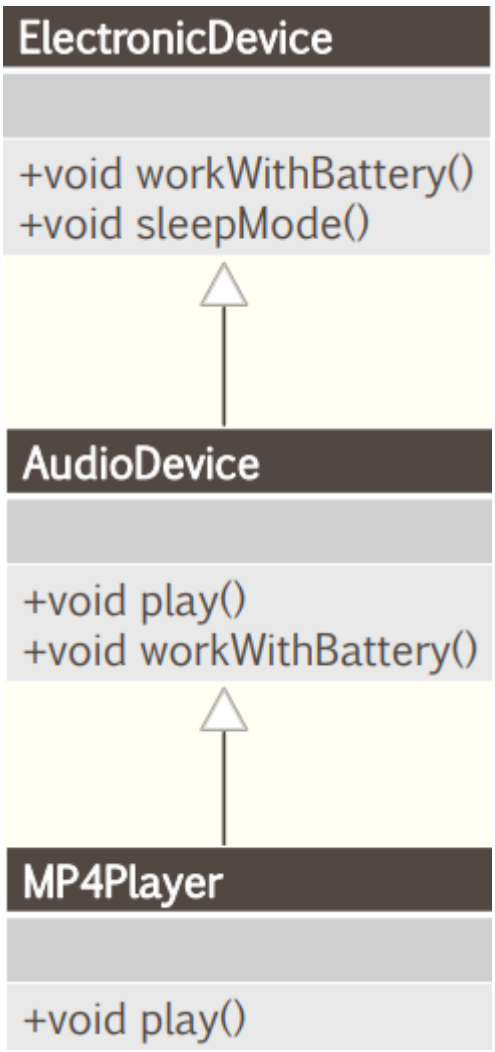
The overridden method has the same method signature (you write the header the same) as the original method. We mark this using the `@override` tag in our code.

You can even call the superclass method using `super.superClassMethod` inside of the overriding method.

If a method defined as `final` then it cannot be overridden.

Subclasses get more and more specific as you go down the inheritance hierarchy.

Example



`AudioDevice` overrides `workWithBatter()`, inherits `sleepMode()`, and adds `play()`.

`MP4Player` overrides play and inherits `workWithBatter()` and `sleepMode()`.

```
MP4Player myPlayer = new MP4Player();
myPlayer.play();
myPlayer.workWithBattery();
myPlayer.sleepMode();
```

When you call a method from a subclass then it assumes that it's the overridden method by default.

How to Recognise an Inheritance Relationship

X IS-A Y means:

- X has all of Y's methods if not more
- X has all the instance variables and attributes that Y has if not more
- Therefore X is a subclass and Y is a superclass

Example:

- A surgeon is-a doctor
  - they have the `doSurgery()` method ig idk
- A square is-a shape
  - they have the `int area` instance variable
- a mountain bike is-a bicycle
  - `canRide(Terrain terrain)` is overridden to behave differently with `Terrain mountain` as an argument

Inheritance and Access Modifiers

Access Modifier	class	Subclass Same Package	Subclass Outside Package	package	World (outside the package)
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	N	N
no access modifier	Y	Y	N	Y	N
private	Y	N	N	N	N

`private` doesn't pass on anything. It's only for that class.

`no access modifier` or `package` passes on for subclasses in the same package and lets others in the package look.

`protected` passes on for all the subclasses.

`public` just lets anyone in on it.

Designing

1. find all the common things
   - what do all of these classes have in common?
2. make the superclass
   - design a class that represents all those common things
3. decide if a subclass needs additional behaviours(methods) or attributes
   - designing subclasses
4. draw the class hierarchy to make sense of what is inherited
   - some features in the superclass should not be inherited

        - make those `private`
    - sometimes things shouldn't be overridden
        - make those `final`
5. sometimes inheritance is used when there is no is-A relationship
    - super class and subclass have many things incommon
    - the source code of superclass is hidden and
        - you need to override its method or add methods/attributes for your needs
        - only make an instance for this purpose if ALL of the inherited things make sense for what you need
    - This is not always a good idea but sometimes there's already code ready for use that you want to take advantage of

Non-Extendable Classes

If a class is defined `final` then it can't be `extends` by a different class.

Access modifier rules still apply when defining classes. If a class is not `public`, classes in different packages can not extend that cass.

If a class only has private constructors then we can't even make an instance of it.

Single Inheritance - Deadly Diamond of Death

Java does not allow classes to extend or inherit from multiple other classes.

If a class has two parents that have different methods of the same name then the subclass won't know which one to inherit.

Inheritance Summary

A subclass:

- extends a superclass
- inherits everything from a superclass
- can override inherited methods
    - lowest overriden method wins the call
- doesn't remove anything
- only has one direct superclass
- can be and do everything the superclass can if not more

Use the is-a test to verify that the inheritance hierarchy is correct

- If ZX extends Y, then X is-a Y must make sense

The IS-A relationship works one way only. A dog is an animal but not all animals are a dog.

The superclass:

- does not know about the existence of any subclass
- can have an unlimited amount of subclasses
- has changes that will effect all of the subclasses

Overriding vs Overloading Methods

To override a method:

- the header/signature should be the exact same
  - same name
  - same arguments
  - same return type
- write `@Override` on top of the method

The access modifier in sub-method should be the same level or with higher access. If we don't do that then the sub-class won't be able to be/do more than the superclass.

To overload a method:

- the name must be the same
- there has to be some difference in the header/signature
  - different argument
    - quantity or type
  - different return type

The access modifier can be anything independent of the super method.

in-class

If you're calling a superclass and a line of code uses the `this` keyword then it will refer to the actual class (the subclass) instead of that superclass.

Queues

A queue is a structure that goes with FIFO (first in first out) principles.

You can only remove elements from the head and add elements to the tail.

A queue can be empty or have a large number of elements in it.

Queues are popular in computing (CPU scheduling algos, WSC Tim Horton's, etc).

It's easier to implement this with a jank ArrayList if you don't want to do it with the actual queue data structure.

It would be better to do this is a has-a relationship as opposed to an is-A relationship.

It's not like a queue is a version of an ArrayList. If we want to treat an ArrayList like a queue we can just encapsulate an ArrayList inside of our own Queue class and only allow the kinds of interactions we want it to have (dequeue or look from the the top, add to the bottom, and get the size).

Even if there's prewritten code you want it might be better to do a has-a relationship instead of is-a. Especially if its an opaque blackbox.

`Object` Class

`Object` is a class that is the ancestor (superclass) of all the classes that are defined in java.

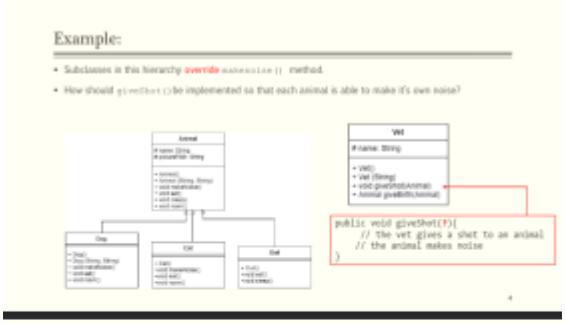
It's where `toString()` comes from. Dastardly, mysterious motherfucker.

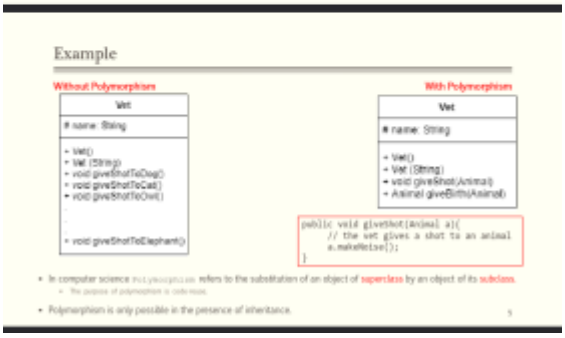
Lect 13: Polymorphism

Polymorphism

Polymorphism is a feature in java that allows us to put an object of a `subclass` instead of an object of that subclass' `superclass`.

You have a method that expects a `superclass` object as an argument but can take an object of a `subclass` of that `superclass` instead.





`giveShotToDog()` only takes the `Dog` type which IS-An `Animal`. But by using polymorphism we can just make a universal `giveShot(Animal a)`.

This works for defining an object:

```
Animal theDog = new Dog();
```

When an object is passed into a method:

```
Vet theVet = new Vet("Jane");
theVet.giveShot(theDog);
```

And when an object is returned from a method:

```
public giveBirth(Animal animal)

Animal puppy1 = theVet.giveBirth(theDog); //this is risky since if we passed in a
different animal type we would run into a runtime error
Dog puppy2 = (Dog) theVet.giveBrith(theDog);
```

`puppy1` is a variable of type `Animal` but points to an object of type `Dog`.

`puppy2` is a variable of type `Dog` and points type `Dog`.

avoiding errors

You can cast a `superclass` object to a `subclass` object. You cannot cast an `child` object to a `sibling` (`child` of the same `parent class`) object.

Suppose this is a new implementation of `giveBirth()`

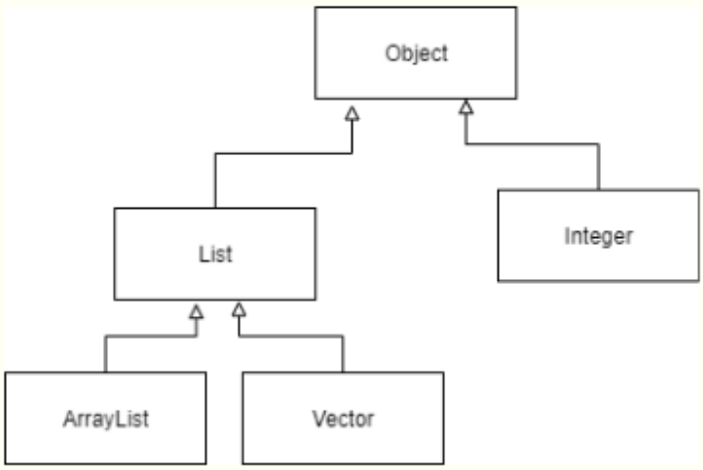
```
public Animal giveBirth(Animal animal) {
  animal.makeNoise();
  Animal baby = new Animal();
  return baby;
}
```

```
Animal puppy = theVet.giveBirth(theDog); //this would be fine

Dog puppy = (Dog) theVet.giveBirth(theDog); //this would be better since we actually have the variable type and object type the same

Cat puppy = (Cat) theVet.giveBirth(theDog); //this would work but it's not what you want
```

Examples

- Which statement is correct?

```
vect.add(0);
array.add(1);
```

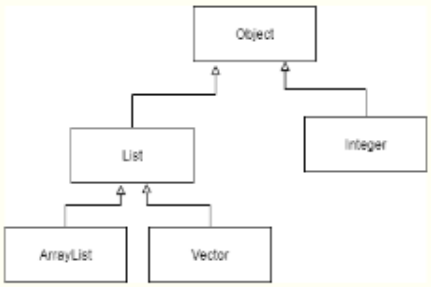



```
List<Integer> vect = new Vector<Integer>();
Object array = new ArrayList<Integer>();
```

`vect.add(0)` is the correct one

- Polymorphism in passing a subtype to a method where the supertype is expected:

```
vect.add((Integer) intValue);
System.out.print(vect.contains(intValue));
```




```
List<Integer> vect = new Vector<Integer>();
Object array = new ArrayList<Integer>();
Object intValue = Integer.getInteger("10");
```

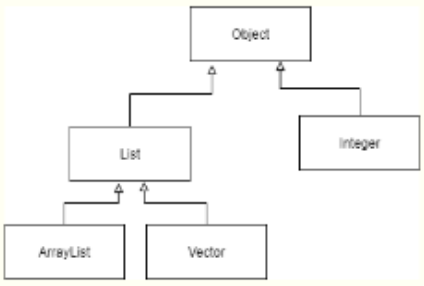
boolean	add(E e)	Appends the specified element to the end of this Vector.
boolean	contains(Object o)	Returns true if this vector contains the specified element.

List has `add()` and `contains()`. `E` is a generic type that will be covered later, we pretend for now that `E` is just `Object`.

When we go to use `vect.contains(intValue)` it would work since we're calling an inherited method inherited from `List` and feeding it an object of type `int` which is a subclass of `Object`

- Polymorphism in returning a subtype where the supertype is expected.

```
List<Integer> vect = new Vector<Integer>();
for (int i = 0; i < 10; i++)
    vect.add(i);

List<Integer> subList = vect.subList(0, 5);
for (int i = 0; i < subList.size(); i++)
    System.out.println(subList.get(i));
```



List<E>	subList(int fromIndex, int toIndex)	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

When we wrote

```
List<Integer> subList = vect.subList(0, 5);
```

`vect.subList(0, 5);` actually returns a `Vector` but it works out since `subList` is a `List`.

Polymorphism Summary

Another principle of oop.

A `subtype` can be presented when the `supertype` is expected.

This is only possible with the inheritance relationship.

With polymorphism the behavior of a method is changed depending on the object that it works on. This is possible do to `Late/Dynamic Binding` which is covered later on.

Phrase borrowed from biology, means "borrowed form".

You can have a variable of `supertype` pointing to a `subtype` (`Animal puppy1` pointing to `Dog`).

But using the `.` you can only get acess to methods defined in `supertype` or overriden in `subtype` but not all new methods in `subtype` .

So if there was a unique method like `rollOver()` then you can't call that using `puppy1`. But you can call `toString()` because every class has that method or `makeNoise()` because it inherited and overwrote it.

This is because `puppy1` thinks it's pointing to an `Animal` instead of a `Dog`.

Lect 14: Static & Dynamic Binding - DBC & Inheritance & Polymorphism

The behaviour of a method changing depending on the object it works on (overriding a superclass' method in the subclass) is possible due to Late/Dynamic binding.

In an inheritance hierarchy there are different versions of a method that may be overridden at different points in the hierarchy. Binding is when Java decides which method definition to use for execution.

Depending on the time that this decision is made we describe it differently

- compile time - early/static binding
- run time - late/dynamic

Dynamic Binding

This occurs when our actual type(supertype) and our declared type(subtype) are different then we call a method that is overridden by the subtype.

```
Shape aShape = new Shape();
Shape firstShape = new Circle(4,  100, 100);
Shape secondShape = new Rectangle(4, 2, 100, 100);
```

```
System.out.println(aShape.toString());
System.out.println(firstShape.toString());
System.out.println(secondShape.toString());
```

Here our declared type is `Shape` but we have three different actual types(`Shape`, `Circle`, and `Rectangle`).

Suppose that later on in our main we use `toString()`. In `Shape`, `toString()` returns "a shape". Contrast to the specific `Shape` subclasses that will return the name of that shape.

Compiling this code we pass through easily, it makes sense to Java since `Shape` has a `toString()` method. At run time Java will look at the actual type and determine that it should use the actual type's method definition for `toString()`.

The definition to be used is determined at run-time, therefore we have dynamic binding.

Static Binding

At compile time the binding between a method call and method definition is determined.

If a method is `final` or `private` it cannot be overridden by the derived classes.

This also occurs with `static`, the `static` method is chosen at compile time. So it would use the `static` definition belonging to the declared type.

DBC - Inheritance & Polymorphism

Recall:

`precondition` is the conditions that need to be true before a method is called in order for the method to work properly.

`postcondition` is the condition that will be true if the `precondition` is true.

`invariant` is a condition that should be true before and after a method is called.

These conditions can be represented with assertions and/or logical expressions and/or other some such.

Inheritance: subclass is-a superclass

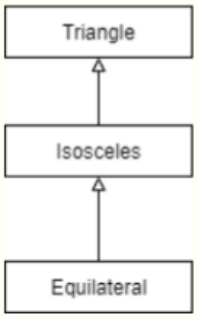
Polymorphism: subclass is-substitutable for a superclass

Whatever a superclass can do, the subclass can do too (even more)

Inheritance & Polymorphism

`invariant`s inherited from the superclass must be either the same or have *more* restrictions than the superclass

- DBC rules for inheritance:
  - The class invariant must not be weaker than the invariant in the superclass
  - Think about the subclass as a more specific type of the superclass.
  - Invariant for Triangle:
    - Sum of angles = 180 degrees
    - Three sides
  - Invariant for Isosceles:
    - Sum of angles = 180 degrees
    - Three sides
    - Two sides have equal length
  - Invariant for Equilateral:
    - Sum of angles = 180 degrees
    - Three sides
    - Three sides have equal length



`precondition`s inherited from the superclass must either be the same or have *less* restrictions. This allows the subclass to do the same things that the super class can do if not even more.

`postcondition`s inherited from the superclass must either be the same or have *more* restrictions than the superclass.

Lect 15: Exceptions

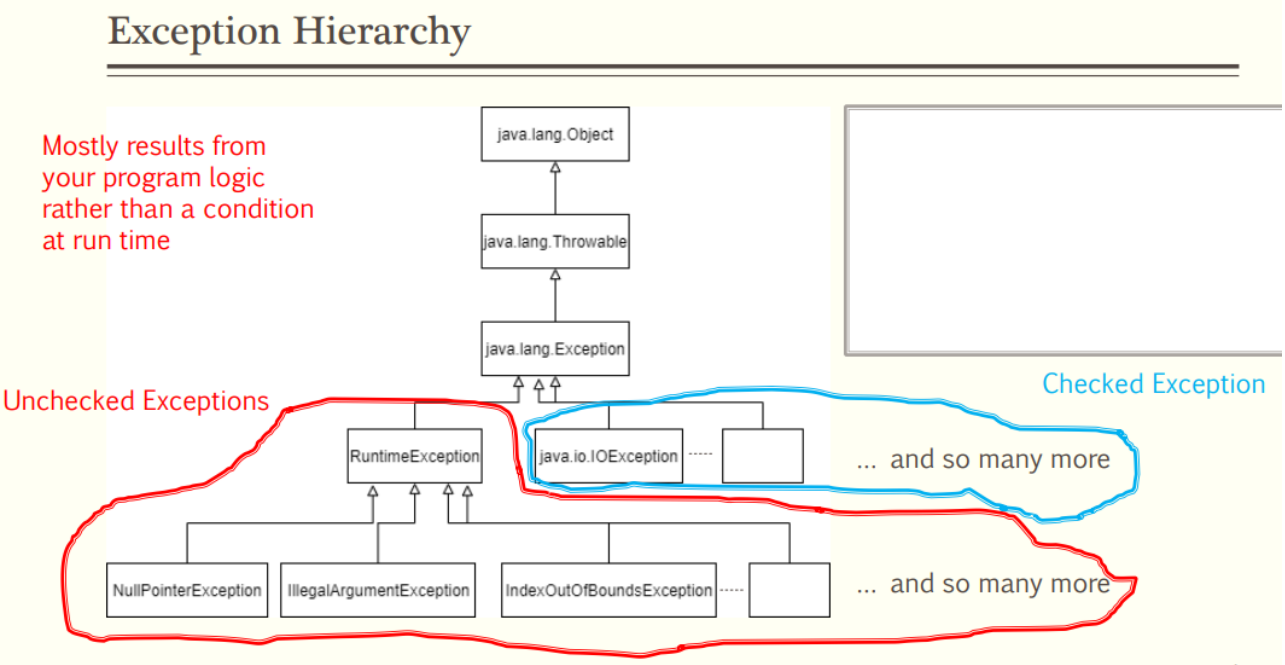
Exceptions

There are two types of exceptions:

1. Unchecked - runtime exceptions
2. Checked - compile time exceptions

We handle unchecked/runtime exceptions by using a throw-catch block

In java `Exception` is a subclass of the `Throwable` class making all exceptions and their subclasses throwable.



What if an Exception happens?

If you handle an unchecked exception using a throw-catch then you can handle it however you want using the catch block.

If you don't handle the unchecked exception then Java will take care of the error.

```
public void wrongMethod1() {
  ArrayList<Integer> arrayObj = new ArrayList<Integer>();
  System.out.println(arrayObj.get(0));
}
```

The above code is an example of an unhandled, unchecked exception.

Java will automatically throw a `IndexOutOfBoundsException` and stop the code from executing.

Checked exceptions are already handled as Java will force you to do it.

How Do We Make Exceptions Help Us?

To make throwing useful we can:

1. handle the exception using a `try-catch`
2. make our own exception that let's us know what's going on

Making An Exception

We can make our own exception to give us more specific information about what's going on.

When making our own `Exception` we `extend` the `Exception` class (an IS-A relationship). We override some of the constructors as well.

In doing this we get all the stack trace related methods (`getStackTrace()`, `printStackTrace()`, `getStackTrace(StrackTraceElement[] stackTrace)`, etc). This shows us precisely where things went wrong and let us have custom messages.

Handling Exceptions

`throw-catch`

```
public void printMonth(int month){
  try {
    if (month < 0) throw new NegativeNumberExcpetion("a negative number is not
accepted as a month!");
    if (month == 0) throw new NumberZeroException("Zero is not accepted as a
month!");
  } catch (Exception e) {
    System.out.println(e.getMessage());
  }
}
```

We can throw as many exceptions as we want, here we throw `NegativeNumberException` and `NumberZeroException`. Both of these extend the `Exception` class so we can catch both of them by just asking for an `Exception` instead of their specific type. They'll both have the `getMessage()` method so we're able to grab their respective messages using dynamic/late binding.

throw without catching

Consider:

```
public void printMonth(int month){
  try {
    if (month < 0) throw new NegativeNumberExcpetion("a negative number is not
accepted as a month!");
    if (month == 0) throw new NumberZeroException("Zero is not accepted as a
month!");
  } catch (Exception e) {
    System.out.println(e.getMessage());
```

```
  }
}
```

and

```
public void printMonth(int month) throws NegativeNumberException,
NumberZeroException{
  try {
    if (month < 0) throw new NegativeNumberExcpetion("a negative number is not
accepted as a month!");
    if (month == 0) throw new NumberZeroException("Zero is not accepted as a
month!");
}
```

The second piece of code doesn't handle the exception within itself and instead throws it to the code that calls it.

User-Defined vs Java Exceptions

Java does not care if you do not handle the native exceptions as it knows how to handle them.

User-defined exceptions (ones that we make ourselves) need to be handled as Java doesn't know how to.

Exceptions and Inheritance

We need to follow the rules of polymorphism which requires that the subclass needs to do the same as the superclass if not more.

If a superclass method throws an exception, the subclass must throw the same or lower level of exception.

Compare:

```
class A {
    public void method1 () throws NumberZeroException{}
    public void method2 () throws NumberZeroException{}
    public void method3 () {}
    public void method4 () throws Exception{}
    public void method5 () throws NumberZeroException{}
}

class B extends A{
    public void method1 () throws NumberZeroException {}
    public void method2 (){}
    public void method3 () throws NumberZeroException{}
    public void method4 () throws NumberZeroException{}
    public void method5 () throws Exception{}
}
```

`method1()`'s override is correct as it's the same.

`method2()`'s override is correct as it imposes less limits.

`method3()`'s override is wrong because it's imposing more limits.

`method4()`'s override is correct as it imposes limits as it only throws a specific kind of limit instead of the whole subclass family of `Exception`.

`method5()`'s override is incorrect as it's the opposite of `method4()`'s.

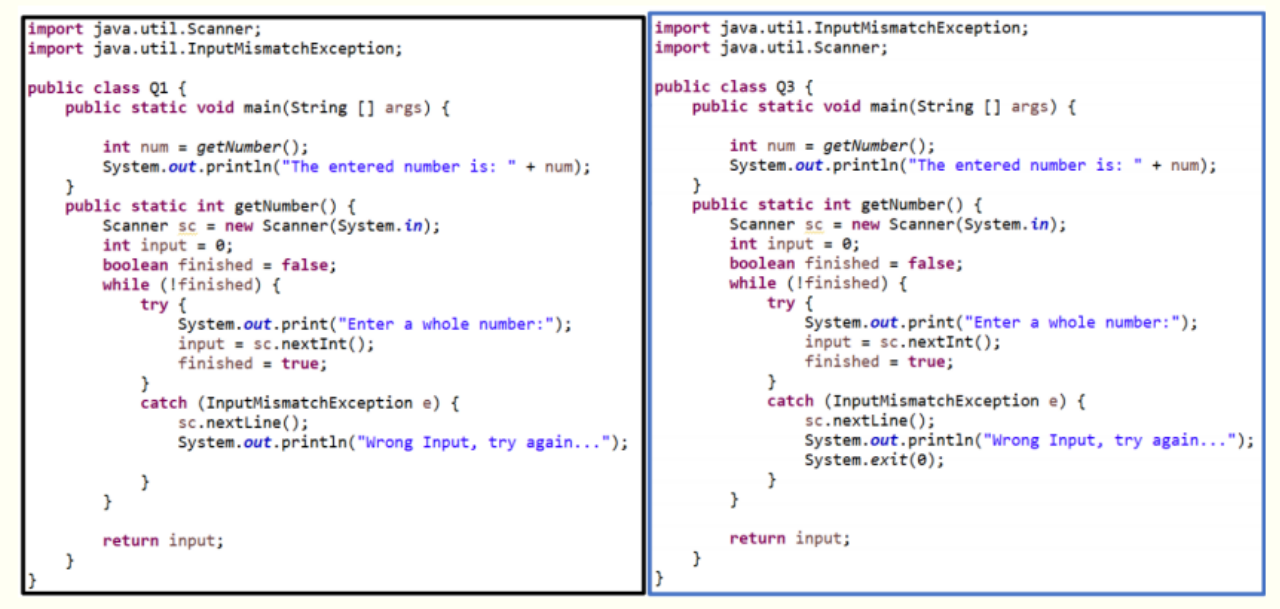
Misc Exception Tips

A method that throws an exception in the method signature can throw any subclass of the exception inside the method.

A method can throw and catch several exceptions.

Exceptions and Program Termination

Handling an exception doesn't necessarily result in program termination.



```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Q1 {
    public static void main(String [] args) {

        int num = getNumber();
        System.out.println("The entered number is: " + num);
    }
    public static int getNumber() {
        Scanner sc = new Scanner(System.in);
        int input = 0;
        boolean finished = false;
        while (!finished) {
            try {
                System.out.print("Enter a whole number:");
                input = sc.nextInt();
                finished = true;
            }
            catch (InputMismatchException e) {
                sc.nextLine();
                System.out.println("Wrong Input, try again...");

            }
        }

        return input;
    }
}
```

```java
import java.util.InputMismatchException;
import java.util.Scanner;

public class Q3 {
    public static void main(String [] args) {

        int num = getNumber();
        System.out.println("The entered number is: " + num);
    }
    public static int getNumber() {
        Scanner sc = new Scanner(System.in);
        int input = 0;
        boolean finished = false;
        while (!finished) {
            try {
                System.out.print("Enter a whole number:");
                input = sc.nextInt();
                finished = true;
            }
            catch (InputMismatchException e) {
                sc.nextLine();
                System.out.println("Wrong Input, try again...");
                System.exit(0);
            }
        }

        return input;
    }
}
```

on the left we just retry until we get a good input from the user, on the right we crash out, either way we're handling the exception.

`Object` Class Cont'd

Obligatory methods (`equals()`, `hashCode()`, `toString()`) need to be overridden to fit our purposes.

`toString`

Retrns the name of the class an `@` then the hashcode of the class in hex (address of the object in the hash table). `className@[hex]`

It's suggested that you override it to make it useful since most of the time one doesn't need the address or class name.

`equals()`

Checks the equality of the object references.

It sees if these references refer to the same memory address, pointing to the same object in the memory.

Lect 16: Obligatory Methods (`Object` class)

Housekeeping

PE1 slight change of grade. (`enoughGate6` was removed).

If you didn't sign the declaration form (declaration that the code you submit it is yours) then you can go to office hours to clear things up. (tue 1600-1700)

Before using the reappraisal form you can try looking at the feedback and using the provided test cases. If it still seems wrong then submit the form.

Term test papers are returned in lab next week.

Obligatory Methods

These are methods that you receive no matter what kind of class or subclass you create.

These methods have a particular implementation that originates from the `Object` class or another class that you're extending.

If the function isn't what you want then you should overwrite it to make it actually useful for you.

`equals()`

`Object`'s `equals()` checks for the equality of `object references`.

Essentially, it checks if two objects are references they check if the references point to the same object. More precisely, it checks the addresses that're stored and sees if they're the same.

It doesn't look at similarities in attributes' values (references or primitives), only the references of the objects themselves.

This isn't always helpful for us so we can overwrite it.

For any non-null reference `x`, `y`, and `z`:

- x.equals(x) -> true
- x.equals(y) -> true iff y.equals(x) -> true

- if x.equals(y) -> true & y.equals(z) -> true then x.equals(z) -> true
- no matter how many times you call `equals()` with two objects it should always be the same
- feeding `equals(null)` should always return false

Overriding

- This example shows you how equals() is implemented, when the class has only primitive attributes

```
public OnlyPrimitives(int first, char second) {
    firstAttribute = first;
    secondAttribute = second;
}
```

```
public boolean equals (Object object) {
    OnlyPrimitives obj = (OnlyPrimitives) object;
    if (firstAttribute == obj.firstAttribute && secondAttribute == obj.secondAttribute)
        return true;
    else return false;
}
```

```
OnlyPrimitives obj1 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj2 = new OnlyPrimitives(10, 'A');
OnlyPrimitives obj3 = obj1;
System.out.println (obj1.equals(obj2));
System.out.println (obj1.equals(obj3));
```

Here we compare all of the attributes from the different objects to see if they're the same and return `true` if they are.

Functions should only have 1 return statement if that so take this example with a grain of salt.

- This example shows you how equals() is implemented, when the class has primitive and simple non-primitive attributes

```
public NonPrimitives(int first, String second) {
    firstAttribute = first;
    secondAttribute = new String(second);
}
```

```
public boolean equals (Object object) {
    NonPrimitives obj = (NonPrimitives) object;
    boolean equal = false;
    if (obj != null)
        if (firstAttribute == obj.firstAttribute && secondAttribute.compareTo(obj.secondAttribute) == 0)
            equal = true;
    return equal;
}
```

```
NonPrimitives obj4 = new NonPrimitives(10, "A");
NonPrimitives obj5 = new NonPrimitives(10, "A");
NonPrimitives obj6 = obj4;
System.out.println (obj4.equals(obj5));
System.out.println (obj4.equals(obj6));
```

Here we do much of the same but for the non-primitive attributes we use `compareTo()`. `compareTo()` in this context will compare the characters and their order in the two strings and see if they're the same. With number objects (`int`, `double`, `float`, etc.) it compares their number value.

If they're the same then `compareTo()` returns `0`, if the object is greater than the other object it returns a number greater than 0, otherwise it returns a number lesser than 0.

Either way, we're comparing the actual value instead of the references by using `compareTo()`.

- This example shows you how equals() is implemented, when the class has complex non-primitive attributes

```
public ComplexNonPrimitives(int first, ArrayList<String> second) {
    firstAttribute = first;
    secondAttribute = new ArrayList<String>();
    for (String obj:second) {
        secondAttribute.add(new String(obj));
    }
}
```

```
public boolean equals (Object object) {
    ComplexNonPrimitives obj = (ComplexNonPrimitives) object;
    boolean equal = (obj != null && this.secondAttribute.size() == obj.secondAttribute.size() &&
                     this.firstAttribute == obj.firstAttribute);
    if (equal)
        for (int i = 0; i < obj.secondAttribute.size(); i++)
            if (secondAttribute.get(i).compareTo(obj.secondAttribute.get(i)) != 0) {
                equal = false;
                break;
            }
    return equal;
}
```

```
ArrayList<String> arr = new ArrayList<String>();
arr.add("A");
arr.add("B");
ComplexNonPrimitives obj7 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj8 = new ComplexNonPrimitives(10, arr);
ComplexNonPrimitives obj9 = obj7;
System.out.println (obj7.equals(obj8));
System.out.println (obj7.equals(obj9));
```

Here we use the above techniques but since we have a complex non-primitive attribute (an array list), we go through all of the attributes within that one and apply `compareTo()`.

If a class we're overriding `equals()` for has another class object as an attribute then we need to know about that other class' `equals()` method in order to implement our own.

Checking Classtype - `getClass()`

Use `getClass()` at the beginning of your `equals()` implementation to check if the two objects are even the same class before trying to compare them and their attributes.

Applications - Searching Techniques

`equals()` makes implementing searching techniques much easier.

Suppose we have a class that has an `ArrayList` and its elements are a class of our own definition.

We can implement `equals()` within that user-defined to help us better search for that object within our `ArrayList`.

`hashCode()`

Hash Tables

This is a data structure that lets us search for an object faster and use `equals()` less in our code.

They're like dictionaries in Python. There are keys and they're associated values that we can retrieve from them in `n(1)` time. The elements are not stored sequentially, we can just retrieve any element and its value in `n(1)` time if we know the key.

In java we have hash tables. There is a bucket array of size `N` and a hash function `h`. We feed the hash function an input, an integer like `k`, and then we get access to the corresponding bucket. `h(k)=k mod N` gives us the location of the object in the bucket array.

`hashCode()`

This method returns an integer. That integer is the memory address in which the object is stored.

Overriding

A hash code must:

- return the same address when frequently calling `obj.hashCode()` at one execution
- obj1.equals(obj2) -> obj1.hashcode () == obj2.hashCode()
    - it is possible for this to fuck up and break but we learn about this in data structures

Overriding is hard so instead we just use `Objects.hash()`, feed it all the attributes, and return the result.

Lect 17: Abstraction & Interfaces

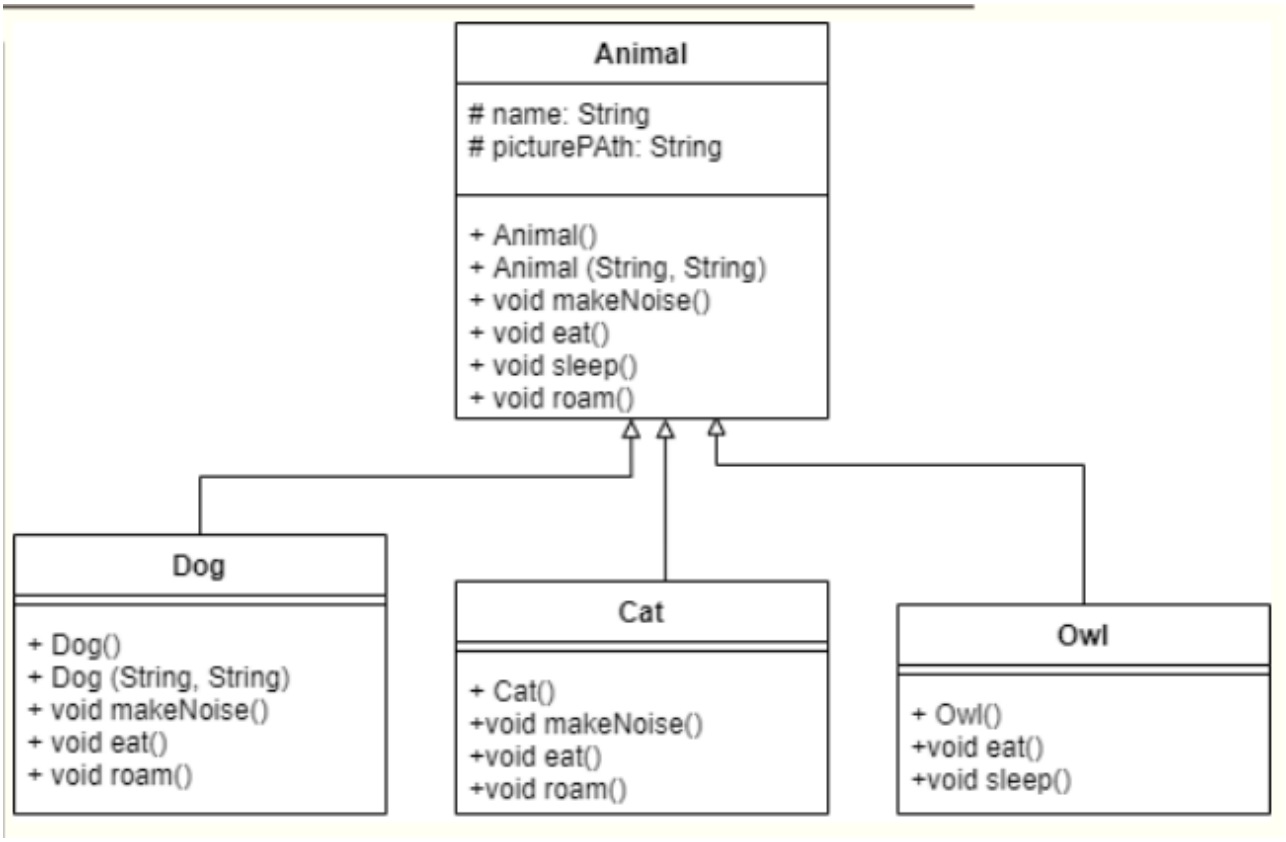
`Comparable` and `compareTo()` were initially covered here but the coverage was so awful I chose to omit it.

Abstract Classes & Methods

Abstract Classes

Some `superclass`es should never be instantiated. Even though they may have a constructor.

Their code is useful and common enough to be made into a `superclass` however they aren't useful on their own.



- an `Animal` class helps us not reuse code for `Dog`, `Cat` or `Owl` but we don't need to instantiate an `Animal` object
  - a general animal isn't specific enough to be useful in most situations but the idea of an animal is useful enough to keep around as an `abstract` concept

These `superclass`es that:

- are never instantiated
- serve more as blueprints for the subclasses than classes in themselves

are called `abstract` classes.

Abstract classes can also inherit other abstract classes.

The `extends` keyword is used to inherit an abstract class.

---

we use the `abstract` keyword after the access modifier and the `class` keyword in the class signature when defining an abstract class.

```
public abstract class Animal {
  public String name;
  public String picturePath;
}
```

```
Animal [] zoo = new Animal [100]; // 1
Animal animal = new Animal();     // 2
Animal herCat = new Cat();        // 3
Dog myPet = new Dog();            // 4
```

Statements 4 and 3 are valid since we aren't making an object of `Animal`.

Statement 2 is invalid since we are attempting to make an object of `Animal`.

Statement 1 is valid as we can set the declared type of the objects of the array as `Animal`, however we don't ever make an object of `Animal` instead filling it subclasses (`Dog`, `Cat`, `Owl`, etc).

This is made possible through polymorphism.

Abstract Methods

Abstract methods are defined by using the `abstract` keyword in the method signature after the access modifier and before the return type.

These methods have no body and have their method signatures instead finished by `;`.

Abstract methods can only exist in abstract classes but abstract classes can have both abstract and non-abstract methods.

```
abstract class Shape {
  int centerX;
  int centerY;
  public final void clearScr() {
    System.out.println("Clear screen is in process....");
    centerX = 0;
    centerY = 0;
  }
// <access modifier> abstract <return type> <method name>();
  public abstract void draw();
```

```
  public abstract double area();
  public abstract double perimeter();
// this method isn't an abstract method so we proceed as normal
  public void moveToCenter() {
    clearScr();
    this.draw();
  }
}
```

All abstract methods should be overridden as soon as they are inherited by a concrete(non-abstract) class.

Summary

Abstract classes help us make full use of the inheritance structure.

Concrete(non-abstract) classes are specific enough such that:

- their attributes get meaningful values
- their methods can be fully implemented

Abstract classes have no meaningful functionality aside from:

- being extended
- used for polymorphism
- for static methods
  - if it has that

Abstract classes must be extended to be useful.

An attempt was made to cover `Interface` in this lecture.

An attempt.

Lect 18: `Interface`, `Comparable`, and `compareTo()`

`Interface`

Suppose we have a pre-existing inheritance hierarchy. We want to add some new functionality to a subclass or specific subclasses but not to other subclasses.

In these cases we make an interface.

There are ways that we could try to get around it but these all have their advantages and disadvantages. ***Feel free to skip this section***

Design 1: add the new functionalities to the superclass.

- +:
  - all relevant subclasses will inherit the new functionalities
  - function will be passed down the hierarchy
  - subclasses can override the method to fir their purpose
  - we can (usually) use polymorphism
- -:
  - subclasses that shouldn't have it now have it

Design 2: add the new functionalities to the superclass and make the superclass abstract.

- +:
  - subclasses that need it get, override, and pass it on
  - we can use polymorphism more often
- -:
  - subclasses that shouldn't have it now have it

Design 3: add the new functionalities to the superclass, make the methods and the superclass abstract.

- +:
  - subclasses that need it get, override, and pass it on
  - we can use polymorphism always
- -:
  - subclasses that shouldn't have it now have it

Design 4: add the new functionalities to the subclasses that they belong to.

- +:
  - subclasses that need it get, override, and pass it on
  - subclasses that shouldn't have it won't have it
- -:
  - we have to write more code
  - sibling classes that fall under the same category will have to follow the same protocol for consistent design
    - an impossible ask
  - can't use polymorphism

Design 5: make an intermediary step in the hierarchy

- +:
  - the specific subclasses we want get the functionality
- -:

    - we're writing a whole new class just to accommodate

Basically:

- if you want to give a subset of subclasses some new functionality then we should use an interface

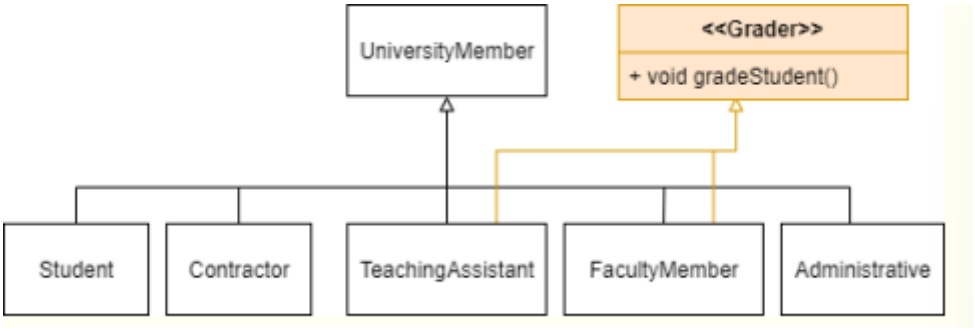
---

Suppose we *still* have that pre-existing inheritance hierarchy. We *still* want to add some new functionality to a subclass or specific subclasses but not to other subclasses.

In these cases we make an interface.

Interface is the way we get around the lack of multiple inheritance in java.

Suppose we still have our pre-existing hierarchy and want to add functionality to a certain subset of the subtypes.



We define an interface using `interface InterfaceName()` similar to how we would define a class. The interface's methods are all `public abstract` even if you don't define it. The subclasses need to implement the actual methods themselves, overriding them in the class' code. Similarly, all of the variables defined in an interface are `public static` implicitly

The header for a class like `FacultyMember` which implements the `<<Grader>>` interface will look something like this:

```
//class <className> extends <superclass> implements <interface1>, <interface2>,
<interface3>, ... , <interfaceN>
class FacultyMember extends UniversityMember implements Grader {
```

The class it extends (is a subclass of) and the interface it implements after.

Note that classes can implement more than one interface Deadly Diamond of Death just doesn't happen :)

In order to ensure proper implementation and usage be sure to:

- make each subclass implement it to fit its purpose
- each subclass will use the same method signature
- only implement it on the subclasses that need it

- to use the interface's methods with polymorphism use the interface as the polymorphic type

  ```
  Grader grader = new TeachingAssistant();
  ```

  With the above code we can call all of the methods within `<<Grader>>` and have the new `TeachingAssistant` do it.

Comparable

Primitives (int, double, float, char, etc.) are easily compared using `< > ==`.

To compare objects (user-defined or otherwise):

- define a way to order them
  - what makes one object larger/higher/better or smaller/lower/worse than another in sorting
- implement a method that compares the two objects in that respect
  - `compareTo()` that we get from `<<Comparable>>`

compareTo()

```
obj1.compareTo(obj2);
```

Returns:

- -ve # when `obj1 < obj2`
- +ve # when `obj1 > obj2`
- 0 when `obj1 = obj2`

`compareTo()` is n `abstract` method that is defined in the `<<Comparable>> Interface`.

```
class myComparableClass implements Comparable {
  public int compareTo(Object obj) {
    // my code
  }
}
```

This method throws 2 exceptions:

- `NullPointerException` - specified object is null
- `ClassCastException` specified object's type prevents it from being compared to this object

For objects `obj1`, `obj2`, `obj3` that are comparable to one another, if:

- `obj1.compareTo(obj2) < 0`
  - then `obj2.compareTo(obj1) > 0`
    - and vice versa
- `obj1.compareTo(obj2) == 0`
  - then `obj2.compareTo(obj1) == 0`
- `obj1.compareTo(obj2) < 0` && `obj2.compareTo(obj3) < 0`
  - then `obj1.compareTo(obj3) < 0`
    - same with `> 0` and `== 0`

Note that if `obj1.cmpareTo(obj2) == 0` then `obj1.equals(obj2)` may or may not be true. They are two different methods and may have different code.

If `obj1.compareTo(obj2) == 0 & obj1.equals(obj2)== true` then we say that `compareTo()` and `equals()` are consistent.

This isn't always the case as `equals()` and `compareTo()` will be consistent with each other. `equals()`, by default, compares the object references while `compareTo()`, varying by implementation, will typically compare instance variables.

Lect 19: Generics

Recall that as programmers we want to rewrite as little code as possible. Generics help us to accomplish this.

An example of where we would want generics would be when we use stacks.

Stacks simple data structures that have a few different methods no matter the type (`push(e)`, `pop()`, `isEmpty()`, and `top()`) and are very popular for a variety of applications (memory stacks, undo history in editing software, compiler understanding parenthesis in your code, tag matching in xml or html files, etc.).

All the different applications will require different versions of a stack in order to accomplish what they set out to do(integer stack, string stack, etc.)

Each different version of stack could be a different implementation that deals with a different datatype or we can make a generic class for all of these to use.

We make a generic stack that we can make an integer, string, etc. type stack when we instantiate it.

Generic Types

Generic types allow us to avoid duplicate code.

A generic type is:

- a(n):
  - interface
  - class
  - method
- whose type is determined by a parameter

In the code we pass the type that we want into the generic type instantiation as an argument then at runtime the actual type is replaced by the argument we fed in.

A normal implementation of an integer stack:

```
class IntegerStack {
  ArrayList<Integer> stack;
  public IntegerStack() {
    stack = new ArrayList<Integer>();
  }
  public void push (int element) {
    stack.add(0, element);
  }
  public int pop() {
    return stack.remove(0);
  }
  public boolean isEmpty() {
    return (stack.size() == 0);
  }
  public int top() {
    return stack.get(0);
  }
}
```

For different implementations of this data structure using a different data type (`String` instead of `int` for example) we would make something similar to the above code except replacing our original data type (int Integer) with the new one (String).

Or we can just make a generic:

```
class Stack <E> {
  ArrayList<E> stack;
  public Stack() {
    stack = new ArrayList<E>();
  }
  public void push (E element) {
    stack.add(0, element);
  }
  public E pop() {
    return stack.remove(0);
  }
```

```
  public boolean isEmpty() {
    return (stack.size() == 0);
  }
  public E top() {
    return stack.get(0);
  }
}

//below is a main function

Stack<Integer> integerStack = new Stack<Integer>();
Stack<String> stringStack = new Stack<String>();
```

In the above code we ued the `<E>` key-phrase to make `Stack` a generic, we could use a different single uppercase letter from `E` as well.

Throughout the class code we use `E` in place of the name of our data type.

When we go to instantiate our generic, `Stack`, we pass in a data type, `Integer` and `String`, in order to make all of the

Generics are very powerful and can change the types of:

- instance variables
- method return types
- parameters/args for methods

only non-primitive data types can replace a generic type

Why?

Generics are built into java in the form of `List<E>` and `ArrayList<E>`.

We've used clever work arounds before where we declared a variable of type `Object` and used casting to make it point to the actual type that we wanted making use of polymorphism.

However when we use workarounds like these it's rather confusing for ourselves and others given that the casting isn't immediately apparent and passing in an incorrect object type will make us run into an error at runtime. Also casting can only go so far but that's neither here nor there.

Generics is more flexible, robust, and easier to read/debug using automated built-in tools and our brains.

We may still use our clever workarounds but when we can help it we should use generics for the reasons above.

Inheritance (1)

```
List<String> stringList = new ArrayList<String>();
stringList.add("some strings");
List<Object> objectList = new ArrayList<Object>();
objectList.add(new Object());
```

```
String str = "a string";
Object obj = str;
```

The above code is fine.

```
List<Object> objectList = stringList;
String s = objectList.get(0);
```

The above code is not fine.

```
String s = objectList.get(0);
```

This will attempt to store an `Object` in a `String` variable.

Even if `type1` is a subclass of `type2` we cannot say that an ArrayList of `type1` is a child of an `ArrayList` of `type2`.

If we try to make an `ArrayList` or any other kind of collection data structure out of the data type we can't say that one is a child of another even if their types have an inheritance hierarchy.

This is an interaction pretty specific to generics.

Array of Generics

```
class GenericsExample<E>{
    E[] array;
    final int ARRAY_SIZE = 10;
    public GenericsExample() {
        array =  new E[ARRAY_SIZE];
    }
    public GenericsExample(E[] input) {
        array = (E[]) new E[input.length];
        for (int i = 0; i < input.length; i++)
            array[i] = input[i];
    }
}
```

`new` wants to reserve space in memory which depends on the object. We don't know how much space a generic type needs to be reserved so we get an error at compile time.

```
class GenericsExample<E>{
    E[] array;
    final int ARRAY_SIZE = 10;
    public GenericsExample() {
        array = (E[])  new Object[ARRAY_SIZE];
    }
    public GenericsExample(E[] input) {
        array = (E[]) new Object[input.length];
        for (int i = 0; i < input.length; i++)
            array[i] = input[i];
    }
}
```

Inheritance (2)

```
String name = "John";
Object object = name;
```

The above code is fine as you can store a subtype in a variable of supertype.

```
String[] names = {"John", "Jane"};
Object[] objects = names;
```

Arrays are said to be `covariant`, which means an array of type `T[ ]`, can contain objects of `S`, where `S` is a subtype of `T`.

Multiple Parameters

A generic can have more than one parameter

- interface `Map<K,V>`
- class `HashMap<K,V>`

```
class Student <K, V> {
  K studentId;
  V studentInfo;
}
```

In the above code we don't know what type `studentId` or `studentInfo` will be. `studentId`

Parameter Naming Conventions

name	use

name	use
E	element
K	key for Map
V	value for Map
N	number
T	“type for” 1st “generic param.”
S	“” 2nd “”
U	“” 3rd “”
V	“” 4th “”

Marzieh wrote `ADT` beside map but didn't explain it. ADT stands for abstract data type. Maps are a thing in different languages, what she wrote is just a reference to that fact essentially.

Generic Methods

Methods can be defined as generics even if the containing class isn't a generic.

```
class GenericsMethodExample{
    public <E> void printMiddle(E[] array){
        System.out.println("Middle elemnet = " + array[array.length/2]);
    }

    public <E> E getMiddle(E[] array){
        return array[array.length/2];
    }
}
```

THe containing class, `GenericsMethodExample`, isn't generic but the methods, `printMiddle()` and `getMiddle()`, are generic.

When we instantiate `GenericsMethodExample` we don't need to specify `E`. Even when we use `printMiddle()` and `getMiddle()` we don't need to specify as it will see the object we passed in as an argument and take its data type to become the new data type.

Wrapper Class

Wrapper classes are static factory methods where we put in a primitive literal and get out an object of that class.

This is used for passing objects into functions that require objects (generics) but this is often handled automatically by java through autoboxing.

Generic Method Cont'd

There are 2 different ways to approach generic methods that we've seen thus far:

- the class is generic and the generic methods rely on the class' generic type
- the class isn't generic and the generic methods rely on their own generic types

We can also make generic methods that rely on their own generic types in addition to the class' generic type or don't use the class' generic type altogether.

Lect 20 has been added before we had it because the lab requires knowledge from it.

This is almost verbatim what's in the autumn-winter 2022 ready-to-wear collection

---

Lect 20: Generics Cont'd

Bounded Type Parameters

Using an inheritance relationship, you can restrict what types are allowed to replace the generic type.

```
public static <T extends Comparable> int counterGreaterThan(T[] array, T element) {
  int count = 0;
  for (T e : array)
    if (e.compareTo(element) > 0)
      count++;
  return count;
```

In the above implementation our generic type `T` is bound to `Comparable`.

Only objects with `Comparable` higher than it in its inheritance hierarchy can be passed into the `counterGreaterThan()` method.

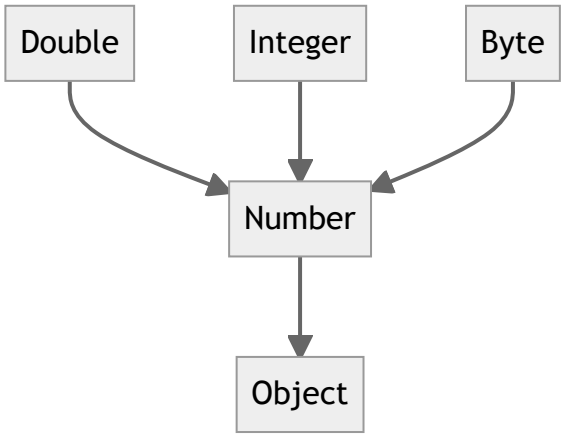
Wildcards

Upper Bound Wildcards

A similar situation would be wanting to allow a class and all of it's subclasses to replace the generic type.

In this case we use an upper bound wildcard, `?`

Take this inheritance hierarchy and code for example.




```
public double sumOfList(List<? extends Number> list) {
  double sum = 0.0;
  for (Number number: list)
    sum += number.doubleVaulue();
  return sum;
}
```

Using the `?` wildcard we're able to except objects of type `Number` as well as objects of types that extend `Number`.

We're able to accept objects of type `Double`, `Integer`, `Byte`, and `Number`.

*Note: simply writing `List<Number>` will not allow us to pass in `List<>`s of a subtype as covered previously in Generic Inheritance

Lower Bound Wildcards

We can even do the inverse.

A situation where we want to allow a class and it's supertypes to be accepted as replacing the generic type.

```
public void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++)
```

```
    list.add(i);
}
```

Here `<? super Integer>` will allow us to pass `Integer` and objects of type that `Integer` extends.

We're able to accept objects of type `Integer`, `Number`, and `Object`.

We can even allow anything using an unbounded wildcard, `<?>`.

This is used when we're either:

- only using `Object`'s methods.
- or when the used methods are independent of type

```
public static boolean equals(List<?> list1, List<?> list2) {
  boolean equal = true;
  if (list1.size() != list2.size()) equal = false;
  else if (list.getClass() != list2.getClass()) equal = false;
  else if (list1 == null || list2 == null) equal = false;
  else
    for (for int i = 0; i < list1.size(); i++)
      if (list1.get(i).equals(list2.get(2))) {
        equal = false;
        break;
      }
  return equal;
}
```

In the above code we used `getClass()` and `equals()` which are `Object`'s methods. We also used `size()` which isn't a method from `Object` but is a part of `List` which we're working with as well.

How To Use Wildcards

Suppose we have a variable who has a generic type.

If it's an:

- IN-variable(producer): a variable whose data is used in the code
    - use upper bound wildcards `<? extends T>`
- OUT-variable(consumer): a variable who receives data in the code
    - use lower bound wildcards `<? Super T>`
- IN/OUT-variable: a variable that gives and receives data in the code

- don't use wildcards

If we the only methods used are from `Object` or independent of type -> unbounded wildcard `<?>`