Week 4

last time: OOP
- building blocks - templates for classes, methods, test classes
  - special methods
  - vars, constants, reserved words, pointers
  - primitive data types
  - strings + custom objects
- statements - arithmetic expressions
  - declarations, assignments, blocks
  - control flow
  - scoping
- JAVA API - String class

today: OOP
- encapsulation + class interactions

outline
part 1b: visibility modifiers
part 2: (Random, Math, Scanner class)
↳ other useful Java classes
part 1a: class relationships

Class relationships

- multiple classes may have various types of reln w. each other:
  - dependency — A uses B
  - aggregation — A has-a B
  - inheritance — A is-a B

  } most common

↳ ch 9, next course

Aggregation

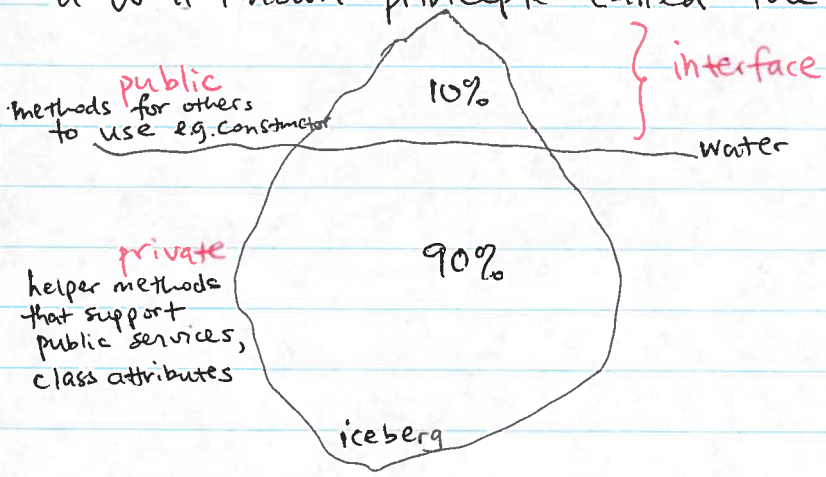
- you've seen lots of examples of this already
  e.g. Game has-a Player
  " has-a Question (in fact, 5 questions)
- in this case A has-a B
  ↳ B is modeled as instance data of A
- here, A relies on B
- that implies: ① if B's design is poor, A's design will be negatively impacted
  ② if B doesn't work properly, A won't either
  ③ if A is faulty, the error can be in B or other parts of A

Dependency

- you've seen several examples of this by now
  e.g. System.out.println
  System.out: object not part of your class attr
  println: service provided
- in this case A uses B
  ↳ B has services (via method calls) that A wants to use
- recall: API - outlines services a class provides to others
- ideally, B offers services to others ← make these publically known
  B should keep everything else hidden ← don't expose your "private parts"

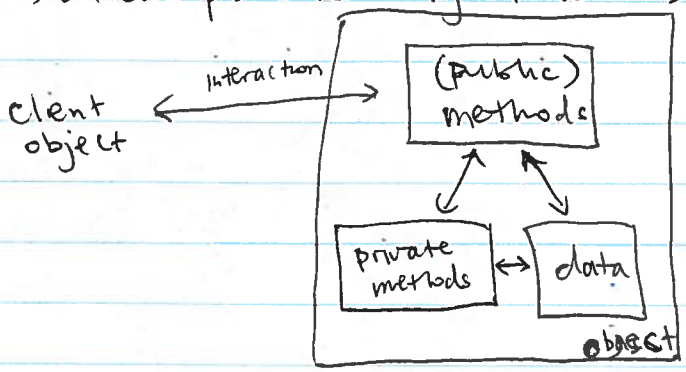
Encapsulation

- viewed internally, an object has variables and methods defined by the class
- viewed externally, an object's methods provide services to other objects in the system
- an object's information should be properly enclosed i.e. encapsulated so other objects cannot see or access info they don't need
- also referred to as information hiding
- a well-known principle called the ice berg principle



- recall: API - gives methods available for use, with some explaination
  - we never see how those methods are implemented

- good practice to keep class attr private ☆
- what if others really need to know their values? accessor
- what if others really need to change their values? mutator
- otherwise, an encapsulated object is a black box



Visibility Modifier

- for now, we consider 2 types of visibility modifiers | private
  public
- need to be applied to: all class attributes
  all method signatures
  all class definitions

BEFORE

```
e.g. class Game
     {
         int numQuestions;
         :
         Game()
         { ...
         }
         boolean isGameOver()
         { ..
         }
         ...
     }
```

NOW

```
public class Game
{
    private int numQuestions;
    :
    public Game()
    { ...
    }
    private boolean isGameOver()
    { ...
    }
    ...
}
```

- why did it work before? default visibility
- when to use which visibility modifier?

default - don't use anymore

private
- all class attributes
- methods that don't need to be used by others
  ↳ how will you know?
- class, if embedded (advanced)

public
- methods that others need to call
- classes you define
- constants → if public, can others change them?

good practise guidelines ☆

Visibility modifiers and encapsulation:

	public	private
class attributes	~~Violate encapsulation~~	Enforce encapsulation
methods	provide services to clients	support other methods within the class

Accessor

- a public method that returns a specific class attribute

e.g.
```
private int numQuestions;
:
public int getNumQuestions()
{
    return numQuestions;
}
```

- also called getter
- template:

```
public [var type] get [var label] ()
{
    return [var attribute];
}
```

Mutator

- a public method that changes the value of a specific class attribute

e.g. private int numQuestions;

```
public void setNumQuestions(int newValue)
{
    numQuestions = newValue;
}
```

- may include error checking in case newValue doesn't fall within valid range
  - don't want program to crash just because a client object doesn't use one service carefully

- also called setter
- template:

```
public void set[var label] ([var type] newValue)
{
    // optional error checking
    [variable] = newValue;
}
```

Part 1 Summary

- class relationships - aggregation | A has-a B |
  - dependency | A uses B |
- encapsulation, information hiding, iceberg principle
- good coding practises
- visibility modifiers - private
  - public
- accessors & mutators

Other Useful Java Classes

① Scanner
② Random
③ Math

the Scanner class

- allows program to be interactive
- takes input from user
  e.g. program asks user question using System.out.println()
  user types in answer
  e.g. program prints out a list of menu options
  user types in selection

- create Scanner object
  e.g.

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

System.in — object for keyboard input

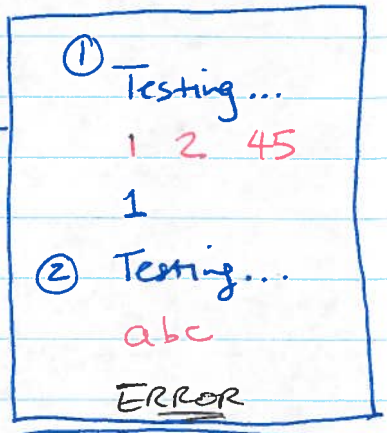
```
:
String userInput;
userInput = sysin.nextLine();
```

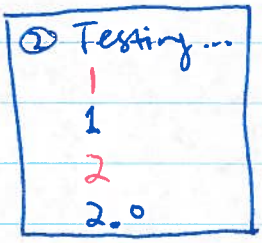
- nextLine() grabs entire line as String
- nextInt() returns user input as int
- nextDouble() " " double

```
S.O.P("Testing...");
System.out.println( sysin.nextLine() );
```

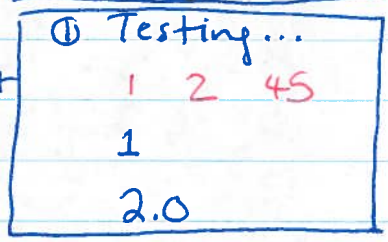
Testing...
1 2 3 456
"1 2 3 456"

- e.g. S.O.P("Testing..."); System.out.println(sysin.nextInt());





- e.g. S.O.P("Testing..."); System.out.println(sysin.nextInt()); System.out.println(sysin.nextDouble());



- to use Scanner class, need at beginning of class file:

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

java.util: name of package; Scanner: name of class

this is called the Import declaration

- by default, every class in java.lang are imported automatically equivalent to having at top of your files:

```
import java.lang.*;
```

* : wildcard, meaning "any" class

- Random class belongs in java.util

- Math class belongs in java.lang