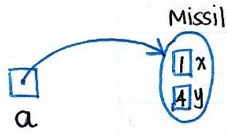
Cheat Sheet : CPSC 210

Object diagrams

□ = field

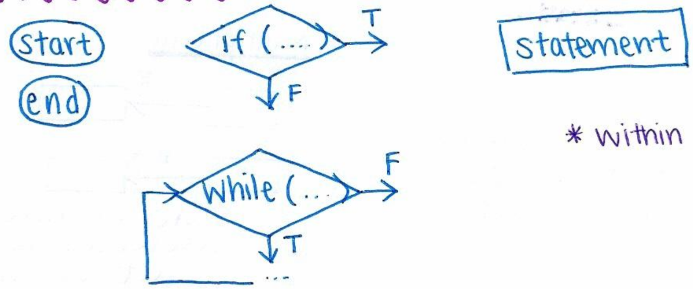
○ = object

fields reference objects.
objects are instances of classes.



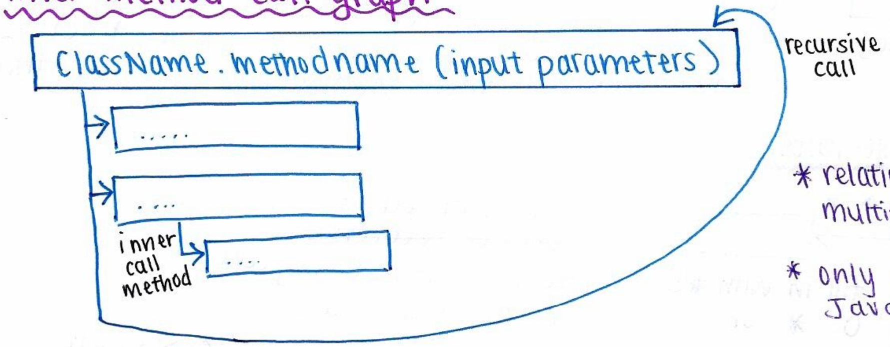
Missile a = new Missile (1,4);

Intra-method flow diagram



* within one method.

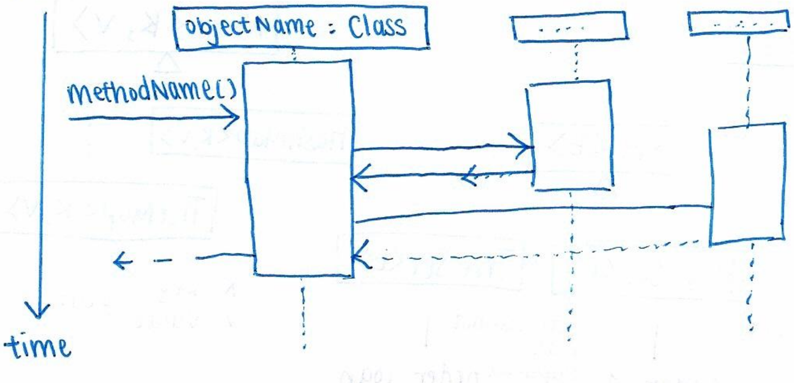
Inter-method call graph



* relationship between multiple methods.

* only methods not in Java library.

UML Sequence Diagram



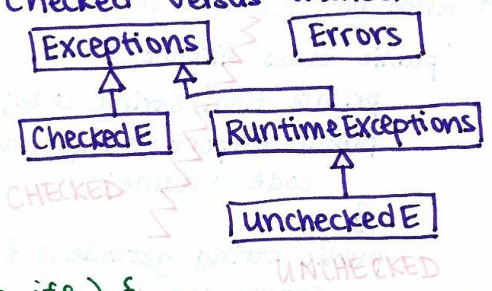
* changing /dynamic relationship

Robustness

- * A class is robust if all of its methods are robust
- * A method is robust if it can handle all input values passed to it.
- * The specified class invariants hold true before and after each method execution.
  - * Remove Requires clause and add throws into Effects clause.
  - * change method header to include throws / or try/catch blocks.

```
public boolean makePayment (...) throws PaymentException {
    if (...) {
        throw new PINException ("...");
    }
    if (...) {
        throw new InsufficientFundsException ("...");
    }
    ....
    }
}
```

If creating exceptions:
Checked versus Unchecked



```
public void charge (...) {
  A ...
    try { makePayment (...);
          C ...
    }
    catch (InsufficientFundsException ife) {
       B System.out.println ("...");
    }
    catch fail ("...");
          D
    }
    finally {
       E ....e.printStackTrace();
              (only for errors/exceptions)
    }
    ...F
}
```

Caught if thrown exception has some type or is a subtype!

If makePayment(...) threw IFE then:

A ✓
C X
B ✓
D X
E ✓
F ✓

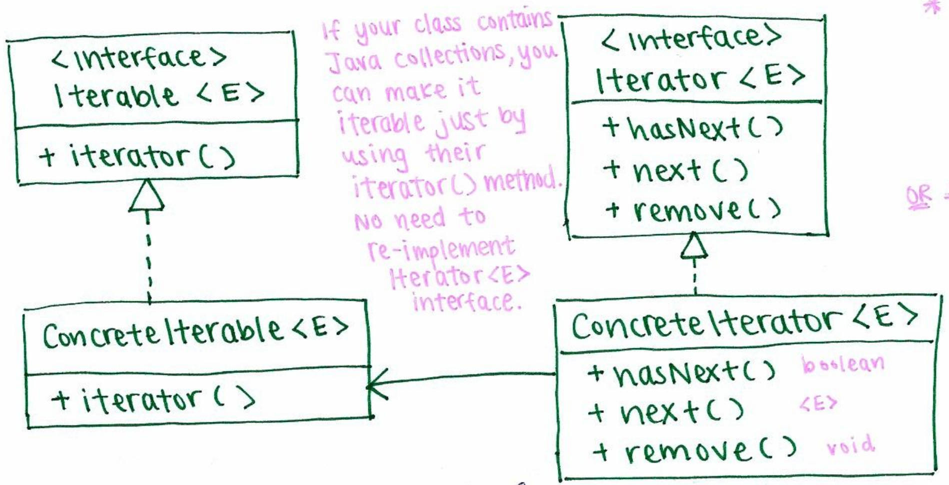
A: always executed
try: execute each method if thrown E, stop!
D/B: if caught, execute inside methods
E: always executed
F: only if no thrown E or if caught and handled.

* Unchecked exceptions do not need to be "thrown" in the method header, and do not need to be caught in a ~~try/block~~ try/catch block.

```
@Test (expected = PaymentException.class)
  public void test throws PaymentException {
  } ...
```

Iteration Pattern

- * provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- * this is done with an Iterator object that knows how to visit all elements in an instance of a collection, then use the Iterator object for all operations no need to know the details of how it is implemented.
- * then use the interface Iterable<E> to provide an Iterator for a collection.



If your class contains Java collections, you can make it iterable just by using their iterator() method. No need to re-implement Iterator<E> interface.

* Disregards the use of while loops, by using for loops as the inner methods do not need to be called upon.

OR → for loop hides the details in a while loop

```
public class IntegerRangeExample {
   private static IntegerRangeExample instance = new IntegerRangeExample();

  public class IntegerRange implements Iterable <Integer> {
       int first, last

       public IntegerRange(int first, int last) {
           this.first = first;
           this.last = last;
       }
      @override
       public Iterator < Integer > iterator() {
            return new IntegerRangeIterator();
        }
```

Iterable

once you do this, can use a for-loop directly on IntegerRange because it is iterable!

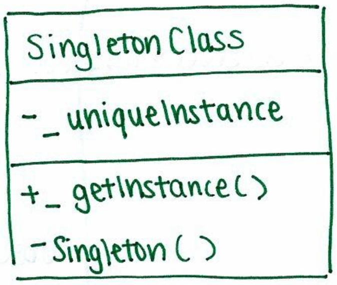
inner class objects can have access to the instances and methods of the enclosing/outer class!

```
    private class IntegerRangeIterator implements Iterator <Integer>{
          private int next = first;
          @override
           public boolean hasNext() {
               return next <= last;
           }
          @override
           public Integer next() {
               Integer result = next;
               next += 1;
               return result;
           }
         @override
          public void remove(){
              throw new
              UnsupportedOperation
              Exception();
          }
    }
```

Iterator

Singleton Pattern

* ensures a class has only one instance - not allowed to create a new instance except through this class
* provides a global point of access to it.



* in place of "new", use the getInstance() method

```
public class Database {
    private static Database uniqueInstance;

    private Database() {
        ...
    }
    public static Database getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Database();
        }
        return uniqueInstance;
    }
}
```

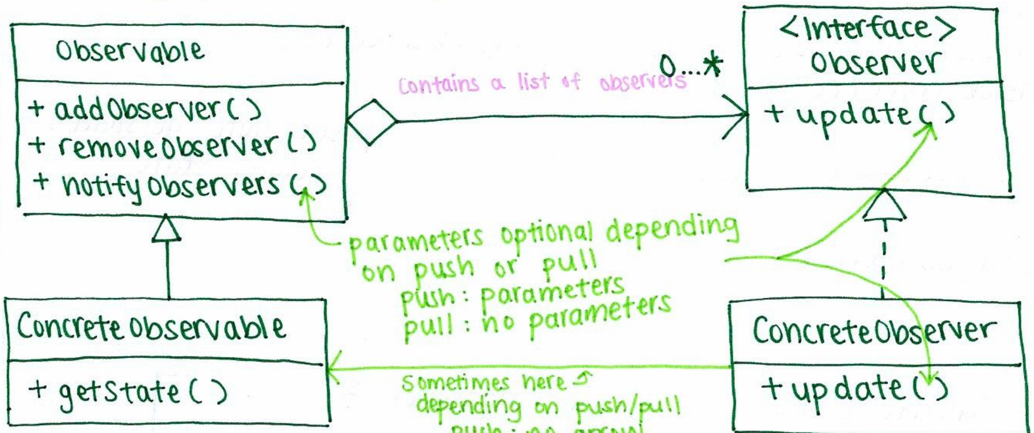
self-reference field →

private constructor →

special one-creation getter method →

Observer Pattern

* defines a one-to-many dependency between objects
* observers are dependent on the observable such that when the observable's state changes, the observers get notified/updated.



parameters optional depending on push or pull
push: parameters
pull: no parameters

sometimes here ↑ depending on push/pull
push: no arrow
pull: yes, getter methods

Benefits:

1. allows for looser coupling: minimize the dependency between objects but they can still interact with each other.
2. can add/remove observers at any time without modifying the observable class
3. able to reuse and expand/evolve without massive changes to the code.

```
public interface Observer {
     public void update ( );
                        ↑ parameter optional
}
```

Observer

```
public class Observable {
     private List<Observer> observers = new ArrayList<Observer>();
     public void addObserver (Observer o) {
           observers.add(o);
      }
     public void removeObserver (Observer o) {
           observers.remove (o);
      }
                                      ↙ parameter optional
     public void notifyObservers ( ) {
          for (Observer o : observers) {
               o.update( );
          }              ↑ parameter optional
     }
}
```

Observable

```
public class ConcreteObservable extends Observable {
     public ConcreteObservable ( ) {
          ...
          ConcreteObserver co = new ConcreteObserver();
          addObserver(co);
     }
     public void/boolean/... operation ( ) {
          ...
          notifyObservers ( );
     }                   ↖ parameter optional
}
```

```
public state getState( ) {
   return state;
}
```

Concrete Observable

```
public class ConcreteObserver implements Observer {
     public void update( ) {...}
                         ↑ parameter optional
```

PUSH (if parameter was given)

```
so update(string s){
      doSomething(s);
   }
```

* do something with the pushed notified state = in this case, a string

PULL (no parameter given)

```
   private ConcreteObservable subject;
   Public void update( ) {
state s = subject.getState();
     doSomething(s);
   }
```

Concrete Observer

Type substitutability

* Can only substitute an instance of the subclass for an instance of a superclass.

* Apparent type = superclass
  Actual type = same as the superclass or its subclass

* The apparent type determines the methods that can be called on the object
* If the actual type provides an implementation for that method and it overrides the super method (or implements the abstract method), it's the one that executes.

Proper / Deep Substitution according to Lisktov's rules

① A subtype cannot strengthen the Requires clause
② A subtype cannot weaken the Effects/Modifies clause
③ A subtype cannot throw more exceptions

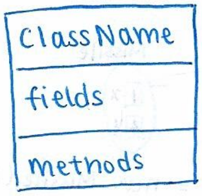
Overriding equals() and hashCode()

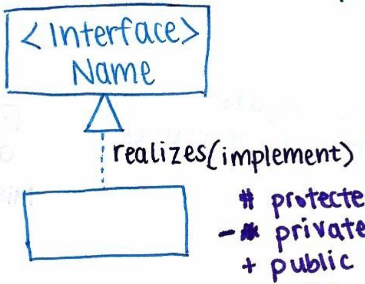
* When you want to use a class object as a key in a hashmap.
* when you want to compare 2 different objects as "equal"

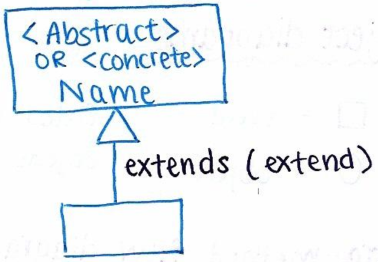
```
public class Airport {
    private final String code;
    public Airport (String name) {
        code = name;
    }
    public String getCode() {
        return code;
    }
   @Override
    public boolean equals (Object o) {
        if (o == null || o.getClass() != this.getClass()) {
            return false;
        }

        Airport other = (Airport) o;
        return other.getCode().equals(this.code);
    }
                          or other.getCode() == this.getCode()
                             if code was a number
  @Override
   public int hashCode() {
       return code.hashCode();
   }
                    or return code if code was a number
}
                    or final int prime = 31;
                       return code.hashCode()* prime + 2code      ← if int
                                                 + 2code.hashCode();
                                                 ↑ if string
```

UML Class Diagram

A class can only extend one superclass but it can implement many interfaces!

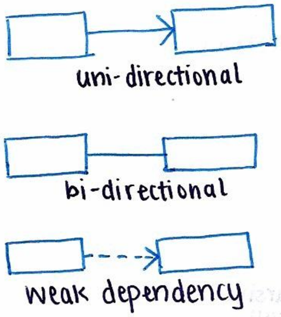






protected
- private
+ public
_ static

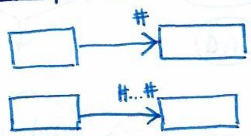
Association / Dependency



stores a reference to another class and vice versa

uses another class object as an input parameter in a method, hence able to access all their methods.

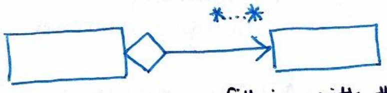
Multiplicity



Stores multiple references to objects of another class.

#...# can be 1...* or 2 or anything.

Aggregation / Whole-Parts

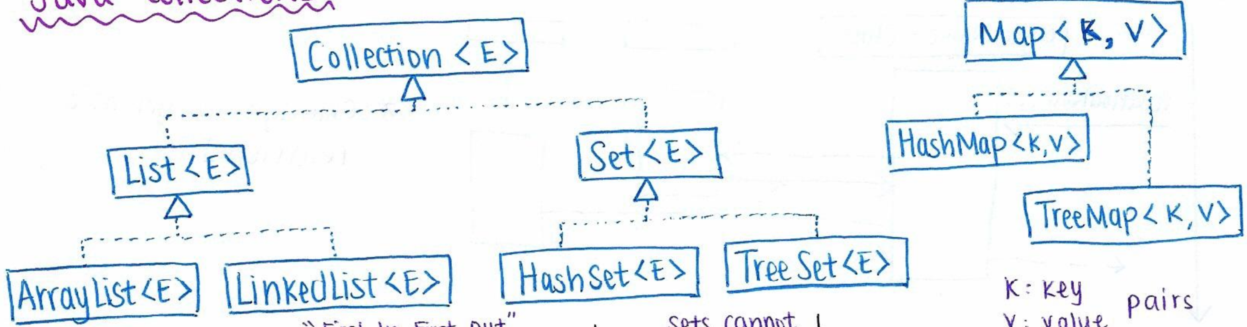


... : fill in with #'s usually 0...* or 1...*

comprised of ~~stores a reference~~ ~~to~~ a list of objects in another class.

example: Album ◇→ Photo

Java Collections



Array Position sequential "First-In-Last-Out"

order n (slowest) O(n)

"First-In-First-Out"

order 1 (fastest) O(1)

Sets cannot have duplicates and the elements are ordered

order log n O(log n)

K: key
V: value pairs