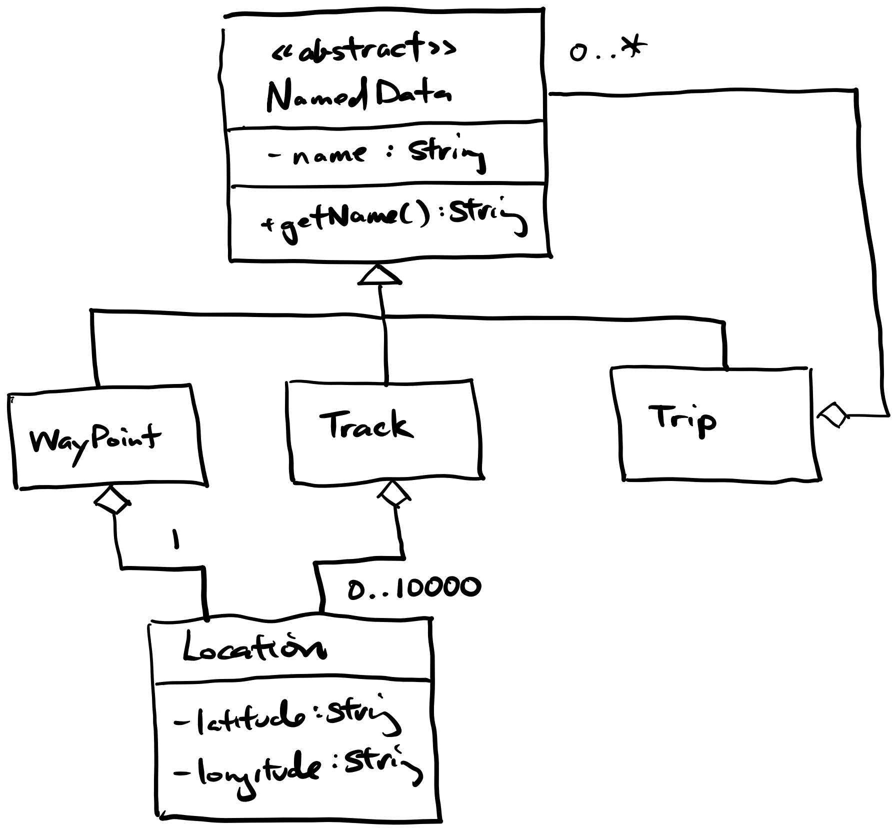
- Object-oriented analysis and design

  (Learning goal: given problem description, draw correct UML class diagram; apply object-oriented design principles.)

  You are designing an application to help manage recorded GPS (Global Positioning System) data. A location has a latitude and longitude. A track is a named sequence of up to 10000 locations. A waypoint is a named location. A trip is a named collection of possibly track(s), waypoint(s), and other trip(s).

[3] Draw a well-designed UML class diagram to represent these entities in the data model for the application. Provide the correct object-oriented abstractions, relationships, attributes, and multiplicities. State any further assumptions, and highlight their appearance in the diagram.


«abstract»
NamedData
- name : String
+getName() : String
0..*
WayPoint
Track
Trip
1
0..10000
Location
-latitude : String
-longitude : String

- Object-oriented analysis and design

  (Learning goal: from/given UML class diagram, write correct Java code)

[2] Accordingly, write correct skeletal Java code for these entities in the data model of the application. Include all abstractions, relationships, attributes, and basic public methods. For example, named entities should have a public `getName` method.

```
import java.util.*;

abstract class NamedData {
    private String name;

    public NamedData( String aName ) {
        name = aName;
    }

    public String getName() {
        return name;
    }
}

class WayPoint extends NamedData {
    private Location location;

    public WayPoint( String aName, Location aLocation ) {
        super( aName );
        location = aLocation;
    }
}

class Track extends NamedData {
    private Location[] locations;

    public Track( String aName ) {
        super( aName );
        locations = new Location[10000];
    }
}

class Trip extends NamedData {
    private ArrayList<NamedData> collection;

    public Trip( String aName ) {
        super( aName );
        collection = new ArrayList<NamedData>();
    }

    public void add( NamedData aNamedData ) {
        if (! collection.contains( aNamedData )) {
            collection.add( aNamedData );
        }
    }
}

class Location {
    private String latitude;
    private String longitude;

    public Location( String aLatitude, String aLongitude ) {
        latitude = aLatitude;
        longitude = aLongitude;
    }
}
```

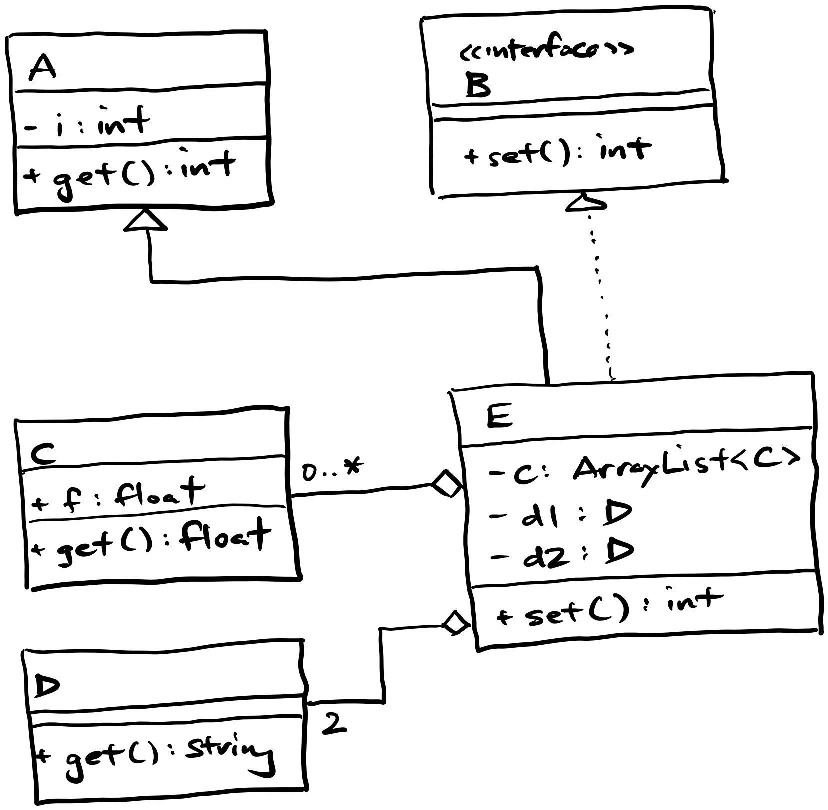
- Object-oriented analysis and design

  (Learning goal: given Java code, draw correct UML class diagram.)

[3] Draw the corresponding UML class diagram for the following skeletal Java code. Provide the correct abstractions, relationships, attributes, methods, and multiplicities.

```
class A {
    private int i;
    public int get() { … }
}

interface B {
    public int set();
}

class C {
    public float f;
    public float get() { … }
}
```

```
class D {
    public String get() { … }
}

public class E extends A implements B {
    private ArrayList<C> c;
    private D d1, d2;

    public int set() { … }
}
```


A
- i : int
+ get() : int
<<interface>>
B
+ set() : int
C
+ f : float
+ get() : float
0..*
E
- c: ArrayList<C>
- d1 : D
- d2 : D
+ set() : int
D
+ get() : String
2

- Object-oriented analysis and design

  (Learning goal: explain whether inheritance is or is not appropriate.)

[2] Suppose a system event log contains events, with just the need to append and iterate over events. A potential software design has corresponding `Log` and `Event` classes. A developer decides to have `Log` inherit from `ArrayList<Event>`. Give two different reasons that clearly explain why this may not be an appropriate software design decision. (Giving an alternative design by itself is not a reason.)

1:

```
A Log is only supposed to allow append and iteration over
events. Conceptually an instance of Log is not
substitutable wherever there is an ArrayList<Event>, so
inheritance is not appropriate here.
```

2:

```
If Log inherits from ArrayList<Event>, it gets endowed with
unnecessary functionality from ArrayList. This inheritance
becomes an assumption by users of Log, making it difficult
to change the data structure later.
```

- Object-oriented analysis and design

  (Learning goal: explain the differences between generalization with inheritance and interfaces.)

[2] Consider abstract classes and interfaces in Java. Explain clearly in what situation(s) should abstract classes be used and in what situations should interfaces be used.

```
Both abstract classes and interfaces can be used to declare
a generalized abstraction or supertype in Java. Subclasses
of abstract classes or implementing classes of interfaces
define a subtype, instances of which can be used wherever
the supertype is expected.

If it makes sense for the supertype to have a partial
implementation, then an abstract class could be used.
However, there's a loss of flexibility, in that once a
subclass inherits from an abstract class, it cannot inherit
from another class, due to single implementation
inheritance in Java.

Since an interface typically doesn't define an
implementation, a class can implement it, along with
possibly other interfaces at the same time more flexibly.
```

- Object-oriented analysis and design

  (Learning goal: explain coupling and cohesion and their relationship.)

[2] Consider coupling and cohesion for the classes of an object-oriented application. Explain clearly how and why these concepts are related. (Defining the terms is not enough.)

```
The structure of an object-oriented application can be
considered like a graph, where the nodes could be classes
and edges could be dependencies among the classes.

One wants a class to be highly cohesive (singular in
purpose), but it likely will then need to depend on other
classes, which leads to high coupling (class dependencies),
and not what we want.

One wants a class to have low coupling, but it likely will
have to do more itself, which leads to low cohesion, also
not what we want.

Generally, there's a tradeoff in trying to achieve both
high cohesion and low coupling at the same time in a
software design.
```

- Software process

  (Learning goal: explain and relate agile manifesto, agile principles, and agile practices.)

[2] The Agile Manifesto describes a principle: “Welcome changing requirements, even late in development.”

From specific Extreme Programming practices, describe two that help to achieve this principle. Explain clearly how. (Just restating the principle or defining the practices is not enough.)

```
In extreme programming, one practice is refactoring,
improving the software design to make it amenable to
accepting new requirements, thus achieving the principle.

Another practice is continuous integration, which supports
small and frequent releases for gaining customer feedback,
thus becoming more welcome to new and change requirements.
```

[2] The Agile Manifesto states as a principle: “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.” From specific Scrum practices, describe two that help to achieve this principle. Explain clearly how. (Just restating the principle or defining the practices is not enough.)

```
In scrum, one practice is the regular sprint review
meeting, where the customer can see their planned and
prioritized user stories being satisfied in a working
software prototype, thus achieving the principle.

Another practice is the sprint planning meeting, where the
customer can help prioritize user stories so that the most
valuable requirements are done early.
```

- Software Process

  (Learning goal: explain and give examples of the differences between incremental and evolutionary prototyping.)

[2] Consider the versions of an operating system that you know. For that system, give a clear example of incremental prototyping and evolutionary prototyping. For each example, explain clearly why it applies a certain type of prototyping. (Do not give examples from the course.)

incremental:

Typically, in incremental prototyping, features are added incrementally through a triage process. Consider the latest macOS (Monterey), where essential features were released at launch, but other features (like Universal Control, which has no priors) were released in a later update.

evolutionary:

In evolutionary prototyping, a feature evolves from a primitive form, to a more refined or robust one. Consider the Mac OS filesystem, which has evolved from MFS (flat structure), HFS (supported directories), HFS+ (supported journaling), and APFS (supported snapshots and crash protection).

- Requirements

  (Learning goals: given problem description, determine a relevant user story; distinguish and give examples of requirement types; for a requirement, provide acceptance tests.)

[4] For a mobile application to help a claimant note expenses for a travel expense claim report, give an example user story for a user requirement and an example user story for a non-functional requirement. For each user story, also provide two acceptance tests for the requirement.

user requirement user story and two acceptance tests:

```
As a claimant, I want to attach receipts digitally with the
expense claim, so that approvers have evidence of my
expenses.

Test that photographs of receipts can be taken with the
camera and attached to an expense claim.

Test that PDF receipts can be attached to an expense claim.
```

non-functional requirement user story and two acceptance tests:

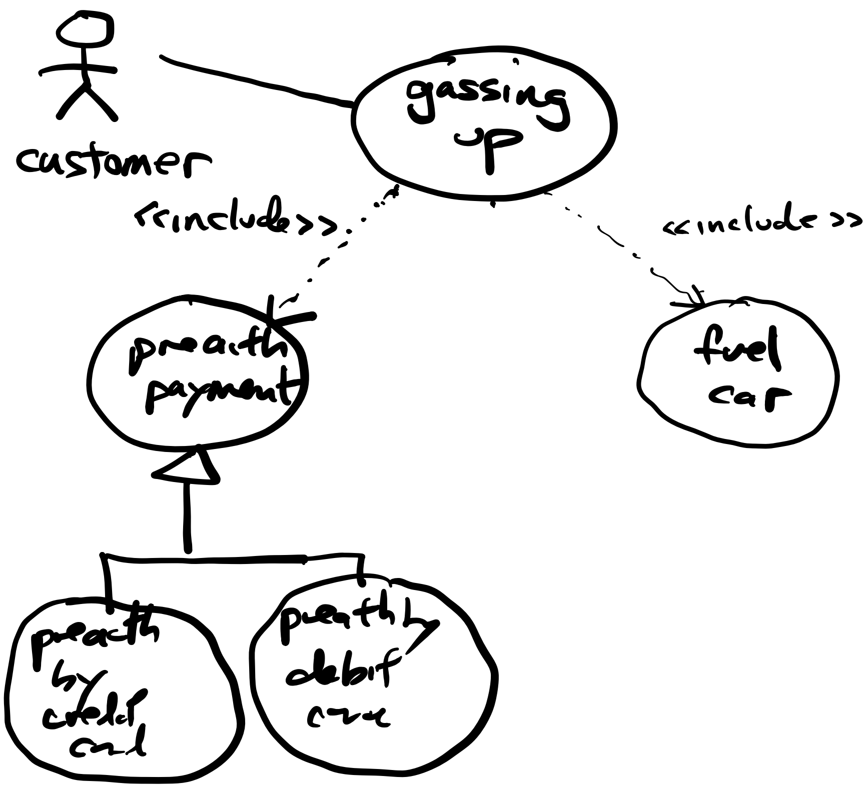
```
As a claimant, I want to submit an expense claim in under 5
seconds, so that I can continue with other tasks.

Test submitting an expense claim with 20 items and attached
digital receipts.

Test submitting an expense claim with 10 items and very
large attached digital receipts.
```

- Requirements

  (Learning goal: given problem description, draw correct UML use case diagram)

  Consider a task with two required subtasks: pre-authorizing payment and fueling at a gas pump at a service station. The pre-authorizing payment subtask has two variations: pre-authorizing by credit card or pre-authorizing by debit card.

[2] Draw the correct UML use case diagram for the task, subtasks, and task variations, showing the actor(s), use cases, and relationships.


customer
gassing up
<<include>>
<<include>>
preauth payment
fuel car
preauth by credit card
preauth by debit card

- Requirements

  (Learning goal: from/given task, write correct use case description.)

[3] Accordingly, write a correct use case description for the fueling subtask, with the following fields. List the steps of what the actor(s) do and what the system presents in the basic flow.

use case name:

```
Fuel Car
```

participating actor(s):

```
Customer
```

goal:

```
To dispense fuel into a car
```

trigger:

```
Customer lifts nozzle
```

precondition:

```
Payment pre-authorized
```

postcondition:

```
Fuel dispensed
```

basic flow:

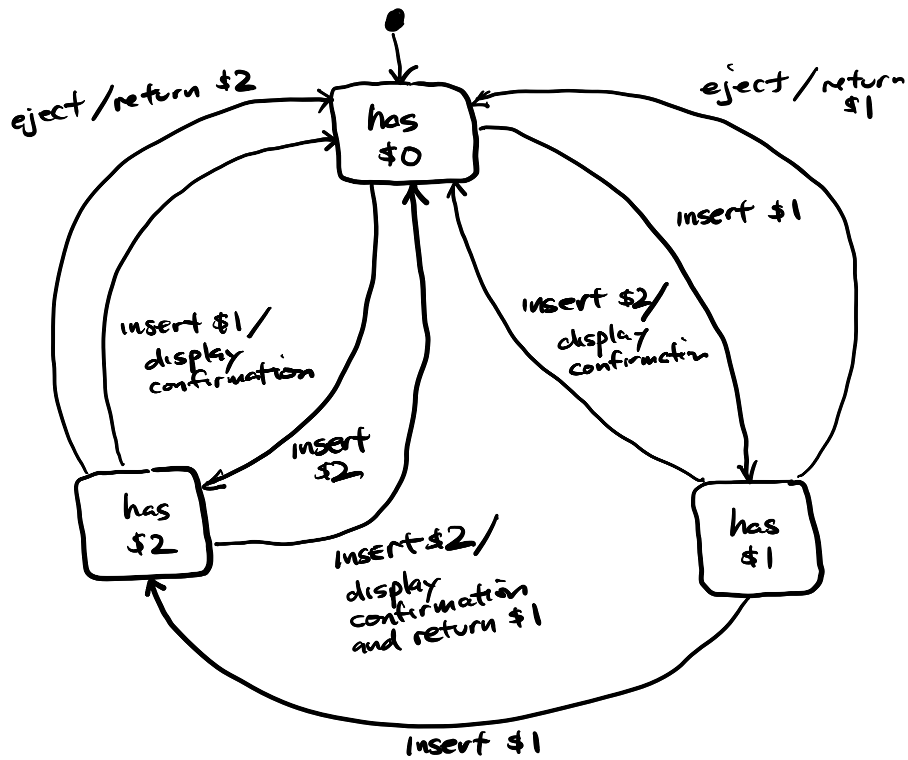
```
1. Customer removes gas cap, lifts nozzle, and inserts
   into car.
2. Pump prompts for fuel type.
3. Customer chooses fuel type, and starts dispensing
   fuel.
4. Pump displays cumulative fuel amount and charge
   continuously.
5. Pump prompts for whether to print receipt.
6. Customer chooses whether to print receipt.
7. Customer stops dispensing fuel, returns nozzle to
   pump, and caps fuel tank.
8. Pump prints receipt if desired.
9. Customer takes receipt if desired.
```

- Requirements

  (Learning goal: given behavior description, draw correct UML state diagram.)

[3] Consider the behavior of a payment machine that accepts only loonies (one dollar coins) and toonies (two dollar coins), one-at-a-time. At least three dollars need to be inserted for the machine to automatically display a confirmation number. The user can also eject the coins inserted so far, if under three dollars. If over three dollars were inserted, the extra amount is returned as change.

For this behavior, draw a correct UML state diagram. Include the relevant states, transitions, triggers, guards, and actions. State any further assumptions, and highlight where each appears in the diagram.


eject / return $2
has $0
eject / return $1
insert $1
insert $1/ display confirmation
insert $2/ display confirmation
insert $2
has $2
insert $2/ display confirmation and return $1
has $1
insert $1