```
"""Simulation Events

This file should contain all of the classes necessary to model the different
kinds of events in the simulation.
"""
from rider import Rider, WAITING, CANCELLED, SATISFIED
from dispatcher import Dispatcher
from driver import Driver
from location import deserialize_location
from monitor import Monitor, RIDER, DRIVER, REQUEST, CANCEL, PICKUP, DROPOFF


class Event:
    """An event.

    Events have an ordering that is based on the event timestamp: Events with
    older timestamps are less than those with newer timestamps.

    This class is abstract; subclasses must implement do().

    You may, if you wish, change the API of this class to add
    extra public methods or attributes. Make sure that anything
    you add makes sense for ALL events, and not just a particular
    event type.

    Document any such changes carefully!

    === Attributes ===
    @type timestamp: int
        A timestamp for this event.
    """

    def __init__(self, timestamp):
        """Initialize an Event with a given timestamp.

        @type self: Event
        @type timestamp: int
            A timestamp for this event.
            Precondition: must be a non-negative integer.
        @rtype: None

        >>> Event(7).timestamp
        7
        """
        self.timestamp = timestamp

    # The following six 'magic methods' are overridden to allow for easy
    # comparison of Event instances. All comparisons simply perform the
    # same comparison on the 'timestamp' attribute of the two events.
    def __eq__(self, other):
        """Return True iff this Event is equal to <other>.

        Two events are equal iff they have the same timestamp.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first == second
        False
        >>> second.timestamp = first.timestamp
        >>> first == second
        True
        """
        return self.timestamp == other.timestamp

    def __ne__(self, other):
        """Return True iff this Event is not equal to <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first != second
        True
        >>> second.timestamp = first.timestamp
        >>> first != second
        False
        """
        return not self == other

    def __lt__(self, other):
        """Return True iff this Event is less than <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first < second
        True
        >>> second < first
        False
        """
        return self.timestamp < other.timestamp

    def __le__(self, other):
        """Return True iff this Event is less than or equal to <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first <= first
        True
        >>> first <= second
        True
        >>> second <= first
        False
        """
        return self.timestamp <= other.timestamp
```

```python
    def __gt__(self, other):
        """Return True iff this Event is greater than <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first > second
        False
        >>> second > first
        True
        """
        return not self <= other

    def __ge__(self, other):
        """Return True iff this Event is greater than or equal to <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first >= first
        True
        >>> first >= second
        False
        >>> second >= first
        True
        """
        return not self < other

    def __str__(self):
        """Return a string representation of this event.

        @type self: Event
        @rtype: str
        """
        raise NotImplementedError("Implemented in a subclass")

    def do(self, dispatcher, monitor):
        """Do this Event.

        Update the state of the simulation, using the dispatcher, and any
        attributes according to the meaning of the event.

        Notify the monitor of any activities that have occurred during the
        event.

        Return a list of new events spawned by this event (making sure the
        timestamps are correct).

        Note: the "business logic" of what actually happens should not be
        handled in any Event classes.

        @type self: Event
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        raise NotImplementedError("Implemented in a subclass")


class RiderRequest(Event):
    """A rider requests a driver.

    === Attributes ===
    @type rider: Rider
        The rider.
    """

    def __init__(self, timestamp, rider):
        """Initialize a RiderRequest event.

        @type self: RiderRequest
        @type rider: Rider
        @rtype: None
        """
        super().__init__(timestamp)
        self.rider = rider

    def do(self, dispatcher, monitor):
        """Assign the rider to a driver or add the rider to a waiting list.
        If the rider is assigned to a driver, the driver starts driving to
        the rider.

        Return a Cancellation event. If the rider is assigned to a driver,
        also return a Pickup event.

        @type self: RiderRequest
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        monitor.notify(self.timestamp, RIDER, REQUEST,
                       self.rider.id, self.rider.origin)

        events = []
        driver = dispatcher.request_driver(self.rider)
        if driver is not None:
            travel_time = driver.start_drive(self.rider.origin)
            events.append(Pickup(self.timestamp + travel_time, self.rider,
                                 driver))
        events.append(Cancellation(self.timestamp + self.rider.patience,
                                   self.rider))
        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: RiderRequest
        @rtype: str
        """
        return "{} -- {}: Request a driver".format(self.timestamp,
                                                    self.rider.id)
```

```
class DriverRequest(Event):
    """A driver requests a rider.

    === Attributes ===
    @type driver: Driver
        The driver.
    """

    def __init__(self, timestamp, driver):
        """Initialize a DriverRequest event.

        @type self: DriverRequest
        @type driver: Driver
        @rtype: None
        """
        super().__init__(timestamp)
        self.driver = driver

    def do(self, dispatcher, monitor):
        """Register the driver, if this is the first request, and
        assign a rider to the driver, if one is available.

        If a rider is available, return a Pickup event.

        @type self: DriverRequest
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        # Notify the monitor about the request.

        # Request a rider from the dispatcher.
        # If there is one available, the driver starts driving towards the
        # rider, and the method returns a Pickup event for when the driver
        # arrives at the riders location.
        monitor.notify(self.timestamp, DRIVER, REQUEST,
                       self.driver.id, self.driver.location)

        events = []
        rider = dispatcher.request_rider(self.driver)
        if rider is not None:
            travel_time = self.driver.start_drive(rider.origin)
            events.append(Pickup(self.timestamp + travel_time, rider,
                                 self.driver))
        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: DriverRequest
        @rtype: str
        """
        return "{} -- {}: Request a rider".format(self.timestamp,
                                                  self.driver.id)


class Cancellation(Event):
    """ The Rider cancels their ride.

    === Attributes ===
    @type rider: Rider
        The rider.

    """
    def __init__(self, timestamp, rider):
        """Initialize a Cancellation event.

        @type self: Cancellation
        @type rider: Rider
        @rtype: None
        """
        super().__init__(timestamp)
        self.rider = rider

    def do(self, dispatcher, monitor):
        """ If the rider hasn't been satisfied, change a waiting rider
            to a cancelled rider and don't schedule any future events.

        @type self: Cancellation
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """

        events = []
        if self.rider.status != SATISFIED:
            dispatcher.cancel_ride(self.rider)
            self.rider.status = CANCELLED
            monitor.notify(self.timestamp, RIDER, CANCEL, self.rider.id,
                           self.rider.origin)

        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: Cancellation
        @rtype: str
        """
        return "{} -- {}: Cancelled their ride.".format(self.timestamp,
                                                        self.rider.id)


class Pickup(Event):
    """ The driver picks up the rider at their origin.

    === Attributes ===
    @type rider: Rider
        The rider.
    @type driver: Driver
        The driver.

    """
    def __init__(self, timestamp, rider, driver):
        """
        @type self: Pickup
        @type rider: Rider
        @type driver: Driver
        @rtype: None
        """

        super().__init__(timestamp)
        self.rider = rider
        self.driver = driver

    def do(self, dispatcher, monitor):
        """ Set the drivers location to the riders location. If the rider is
            waiting, the driver begins giving a ride to the rider and the
            drivers destination becomes the riders destination and return a
            Dropoff event.

            If the rider has cancelled, return a DriverRequest event.

        @type self: Pickup
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """

        events = []
        self.driver.end_drive()
        if self.rider.status == WAITING:
            monitor.notify(self.timestamp, RIDER, PICKUP, self.rider.id,
                           self.rider.origin)
            monitor.notify(self.timestamp, DRIVER, PICKUP, self.driver.id,
                           self.driver.location)
            travel_time = self.driver.start_ride(self.rider)
            events.append(Dropoff(self.timestamp + travel_time, self.rider,
                                  self.driver))
            self.rider.status = SATISFIED
        elif self.rider.status == CANCELLED:
            events.append(DriverRequest(self.timestamp, self.driver))

        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: Pickup
        @rtype: str
        """
        return "{} -- {}: Picked up {}".format(self.timestamp, self.driver.id,
                                               self.rider.id)


class Dropoff(Event):
    """ The driver drops of the rider at their destination.

    === Attributes ===
    @type rider: Rider
        The rider.
    @type driver: Driver
        The driver.
    """

    def __init__(self, timestamp, rider, driver):
        """

        @type self: Dropoff
        @type rider: Rider
        @type driver: Driver
        @rtype: None
        """

        super().__init__(timestamp)
        self.rider = rider
        self.driver = driver

    def do(self, dispatcher, monitor):
        """ Set the drivers location to the riders destination. Return a
            DriverRequest event. The driver has no destination.

        @type self: Dropoff
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """

        events = []
        self.driver.end_ride()
        monitor.notify(self.timestamp, DRIVER, DROPOFF,
                       self.driver.id, self.driver.location)
        events.append(DriverRequest(self.timestamp, self.driver))

        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: Dropoff
        @rtype: str
        """
        return "{} -- {}: Dropped off {}".format(self.timestamp, self.driver.id,
                                                 self.rider.id)


def create_event_list(filename):
    """Return a list of Events based on raw list of events in <filename>.

    Precondition: the file stored at <filename> is in the format specified
    by the assignment handout.

    @param filename: str
        The name of a file that contains the list of events.
    @rtype: list[Event]
    """
    events = []
    with open(filename, "r") as file:
        for line in file:
            line = line.strip()

            if not line or line.startswith("#"):
                # Skip lines that are blank or start with #.
                continue

            # Create a list of words in the line, e.g.
            # ['10', 'RiderRequest', 'Cerise', '4,2', '1,5', '15'].
            # Note that these are strings, and you'll need to convert some
            # of them to a different type.
            tokens = line.split()
            time_stamp = int(tokens[0])
            event_type = tokens[1]

            # HINT: Use Location.deserialize to convert the location string to
            # a location.

            if event_type == "DriverRequest":
                # Create a DriverRequest event.
                identity = tokens[2]
                location = deserialize_location(tokens[3])
                speed = int(tokens[4])
                driver = Driver(identity, location, speed)
                event = DriverRequest(time_stamp, driver)

            elif event_type == "RiderRequest":
                # Create a RiderRequest event.
                identity = tokens[2]
                origin = deserialize_location(tokens[3])
                destination = deserialize_location(tokens[4])
                patience = int(tokens[5])
                rider = Rider(identity, origin, destination, patience)
                event = RiderRequest(time_stamp, rider)

            events.append(event)

    return events
```