Module 21

Object-Oriented Design

What Do We Mean by OO Design?

- Remember how we learned about functions?
  - First learned to **call** functions made for us
  - Then learned to **define** our functions
  - Finally learned to properly **design** functions
- We are following the same path for classes
  - First learned how to **instantiate** classes
  - Then learned to **define** our own classes
  - Now it is time to learn how to **design** classes

Object Oriented Design

Interface

- How the code fits together
  - interface btw programmers
  - interface btw parts of an app
- Given by **specifications**
  - Class spec and invariants
  - Method specs and preconds
  - Interface is **ALL of these**

Implementation

- What the code actually does
  - when create an object
  - when call a method
- Given by method **definitions**
  - Must meet specifications
  - Must not violate invariants
  - But otherwise flexible

Important concept for making large software systems

Interface vs Implementation

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
    Inv: min is an int in 0..59"""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        self.hour = hour
        self.min = min

    def increment(self, hours, mins):
        """Move time hours, mins in future
        Pre: hours int >= 0; mins in 0..59"""
        self.hours += hours + (mins//60)
        self.mins += mins % 60
```

Interface

Interface

Implementation

Interface

Implementation

Designing Types

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, //, …)
  - Time (**set**: times of day; **ops**: time span, before/after, …)
  - Worker (**set**: all possible workers; **ops**: hire,pay,promote,…)
  - Rectangle (**set**: all axis-aligned rectangles in 2D;
    ops: contains, intersect, …)
- To define a class, think of a *real type* you want to make
  - Python gives you the tools, but does not do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want

Making a Class into a Type

1. Think about what values you want in the set
   - What are the attributes? What values can they have?
2. Think about what operations you want
   - This often influences the previous question

- To make (1) precise: write a *class invariant*
  - Statement we promise to keep true **after every method call**
- To make (2) precise: write *method specifications*
  - Statement of what method does/what it expects (preconditions)
- Write your code to make these statements true!

Planning out a Class

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
    Inv: min is an int in 0..59"""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""

    def increment(self, hours, mins):
        """Move time hours and mins
        into the future.
        Pre: hours int >= 0; mins in 0..59"""

    def isPM(self):
        """Returns: True if noon or later."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Time instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Rectangle(object):
    """Class to represent rectangular region

    Inv: t (top edge) is a float
    Inv: l (left edge) is a float
    Inv: b (bottom edge) is a float
    Inv: r (right edge) is a float
    Additional Inv: l <= r and b <= t."""

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b]
        Pre: args are floats; l <= r; b <= t"""

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
           intersection of self with other."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Rectangle(object):
    """Class to represent rectangular region

    Inv: t (top edge) is a float
    Inv: l (left edge) is a float
    Inv: b (bottom edge) is a float
    Inv: r (right edge) is a float
    Additional Inv: l <= r and b <= t."""

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b
        Pre: args are floats; l <= r; l

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
           intersection of self with other."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Special invariant **relating** attributes to each other

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Hand(object):
    """Instances represent a hand in cards.

    Inv: cards is a list of Card objects.
    This list is sorted according to the
    ordering defined by the Card class."""

    def __init__(self, deck, n):
        """Draw a hand of n cards.
        Pre: deck is a list of >= n cards"""

    def isFullHouse(self):
        """Return: True if this hand is a full
        house; False otherwise"""

    def discard(self, k):
        """Discard the k-th card."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

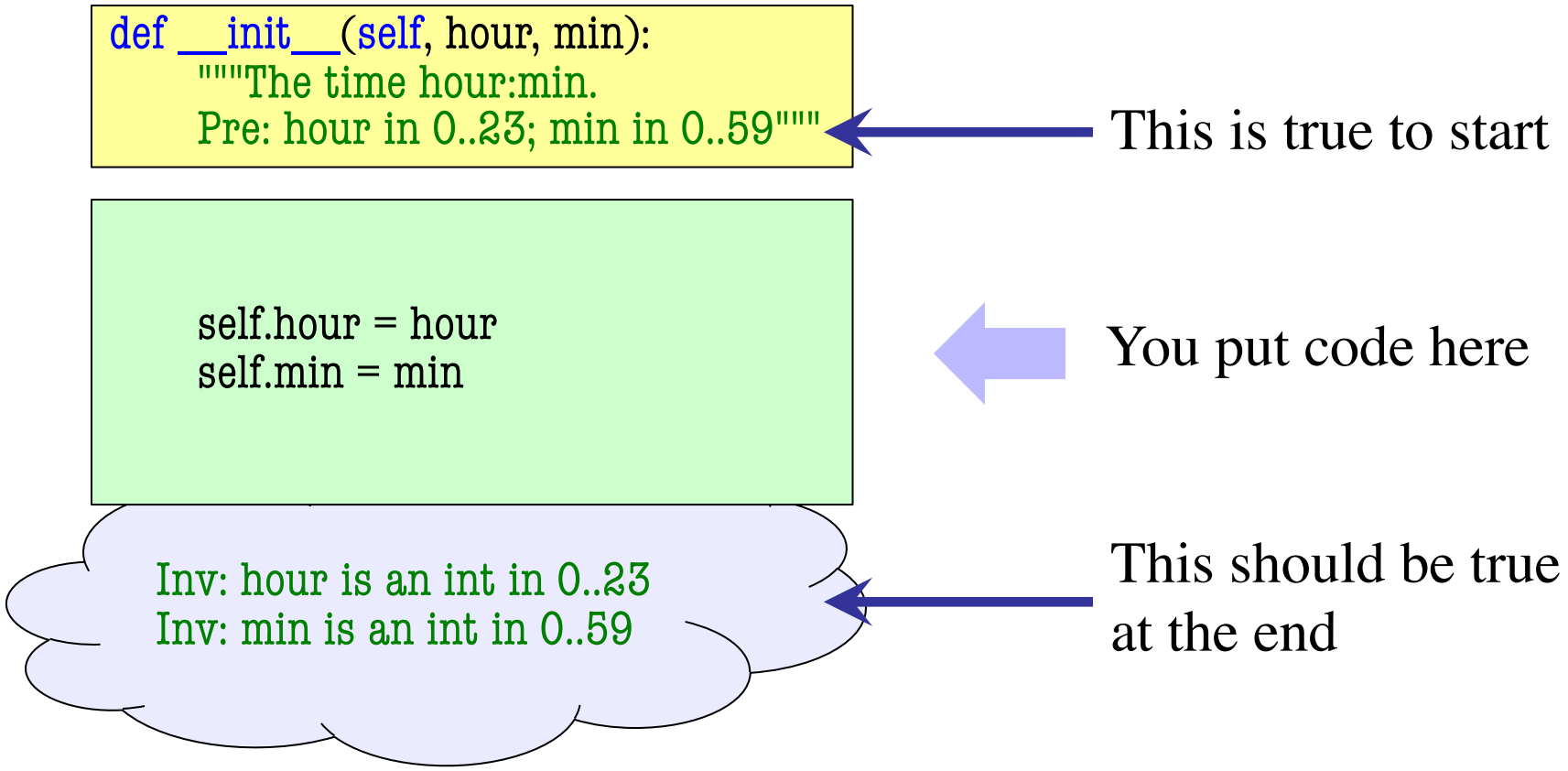
States what the method does.

Gives preconditions stating what is assumed true of the arguments.

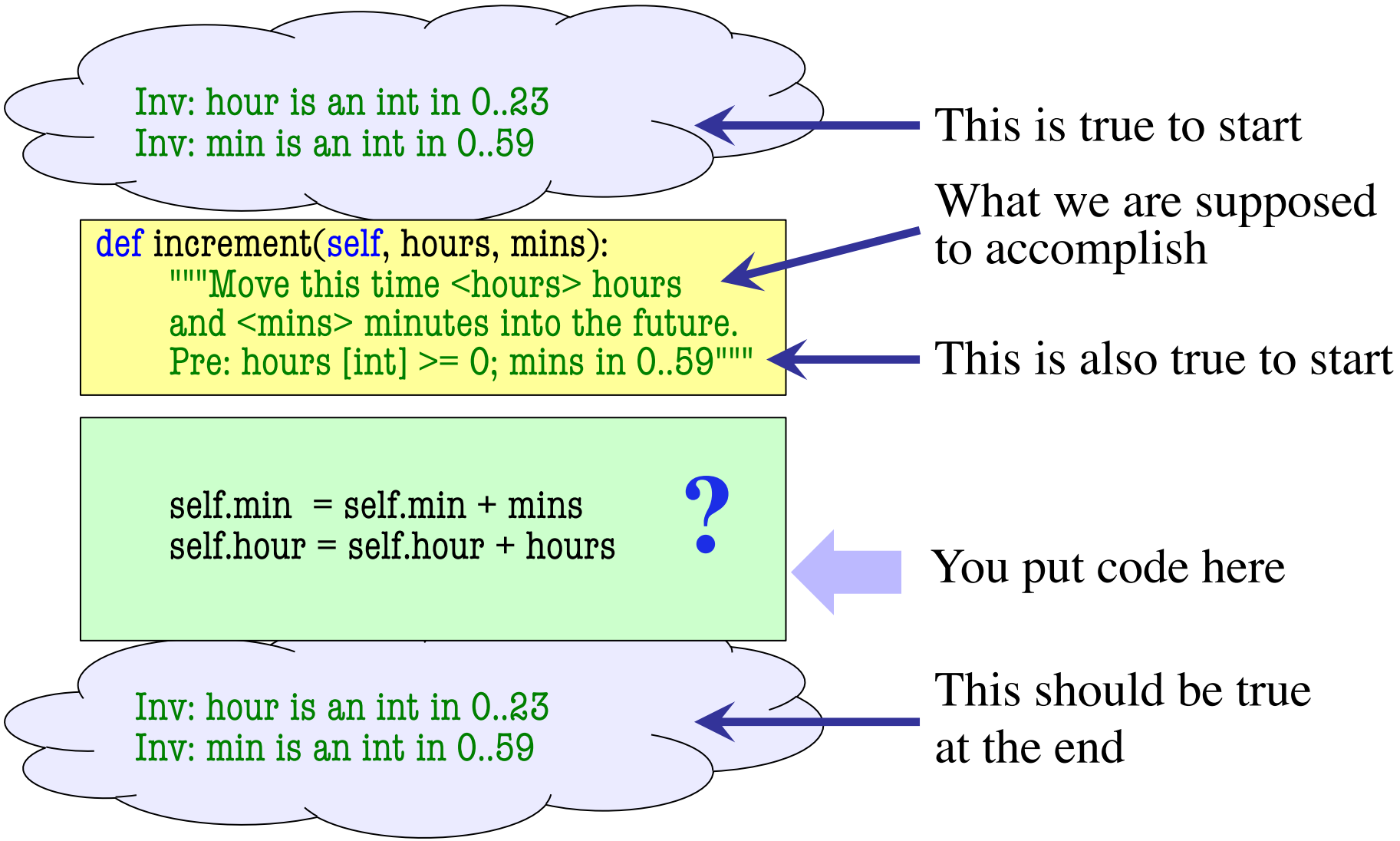
Implementing a Class

- All that remains is to fill in the methods. (All?!)
- When **implementing methods**:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when **using the class**:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

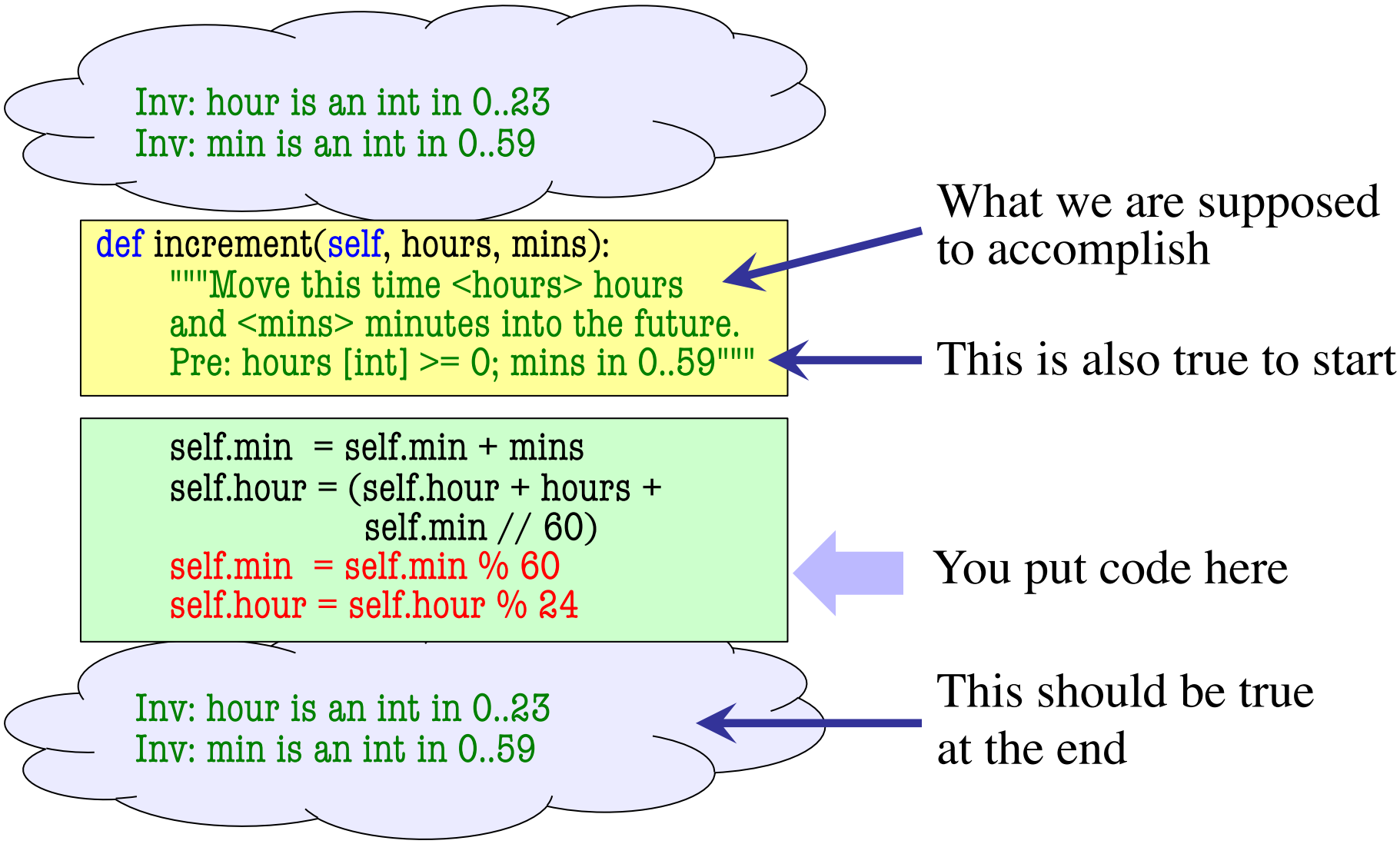
Implementing an Initializer



Implementing a Method



Implementing a Method



Implementing a Class

- All that remains is to fill in the methods. (All?!)
- When **implementing methods**:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when **using the class**:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Recall: Enforce Preconditions with assert

```
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: n an int, 0 < n < 1,000,000"""
    assert type(n) == int, str(n)+' is not an int'
    assert 0 < n and n < 1000000, repr(n)+' is out of range'
    # Implement method here...
```

Check (part of) the precondition

(Optional) Error message when precondition violated

Enforce Method Preconditions with assert

```
class Time(object):
    """Class to represent times of day."""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""
        assert type(hour) == int
        assert type (min) == int
        assert hour >= 0
        assert 0 <= min and min < 60
```

Inv: hour is an int in 0..23
Inv: min is an int in 0..59"""

Initializer creates/initializes all of the instance attributes.

Asserts in initializer guarantee the initial values satisfy the invariant.

Asserts in other methods enforce the method preconditions.

Enforcing Invariants

```
class Time(object):
    """Class to repr times of day.

    Inv: hour is an int in 0..23
    Inv: min is an int in 0..59
    """
```

Invariants: Properties that are always true.

- These are just comments!

```
>>> t = Time(2,30)
>>> t.hour = 'Hello'
```

- How do we prevent this?

- **Idea**: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- **Example**:

```
def getHour(self):
    """Returns: the hour"""
    return self.hour

def setHour (self,value):
    """Sets hour to value"""
    assert type(value) == int
    assert value >= 0 and value < 24
    self.numerator = value
```

Setters and Getters

Setter Method

- Used to change attribute
- Replaces all assignment statements to the attribute
- **Bad**:

```
>>> t.hour = 5
```

- **Good**:

```
>>> t.setHour(5)
```

Getter Method

- Used to access attribute
- Replaces all usage of attribute in an expression
- **Bad**:

```
>>> x = 3*t.hour
```

- **Good**:

```
>>> x = 3*t.getHour()
```

Setters and Getters

Setter Method

- Used to change attribute
- Replaces all assignment statements to the attribute
- **Ba**
  - >>
- **G**
  - >>

Restricts Intentional Changes

Getter Method

- Used to access attribute
- Replaces all usage of attribute in an expression
- **Ba**
  - >>
- **G**
  - >>

Restricts Accidental Changes

The Problem with Getters/Setters

- **Idea**: Force the user to only use methods
  - Do not allow direct access of attributes
- But what is stopping direct access?
  - Attributes are still there! Methods
  - In fact, mentioned in class invariant
- We want **data encapsulation**
  - Make impossible (or nearly) for direct access
  - User only knows to access through methods

Hiding Methods From Access

- Hidden methods
  - start with an **underscore**
  - do not show up in **help()**
  - are meant to be **internal** (e.g. helper methods)
- But they are **not restricted**
  - You can still access them
  - But this is bad practice!
  - Like a precond violation
- Can do same for attributes
  - Underscore makes it hidden
  - Only used inside of methods

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
    Inv: min is an int in 0..59"""

    def _is_minute(self,m):
        """Return: True if m valid minute"""
        return (type(m) == int and
                m >= 0 and m < 60)

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        assert self._is_minute(m)
        ...
```

Helper

Hiding Methods From Access

- Hidden methods
  - start with an **underscore**
  - do not show up in **help()**
  - are meant to be **internal** (e.g. helper methods)
- But they are **not restricted**
  - You can still access them
  - But this is bad practice!
  - Like a precond violation
- Can do same for attributes
  - Underscore makes it hidden
  - Only used inside of methods

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
         min is an int in 0..59"""

    def _is_minute(self,m):
        """Return: True if m valid minute"""
        return (type(m) == int and
                m >= 0 and m < 60)

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        assert self._is_minute(m)
        ...
```

HIDDEN

Helper

Data Encapsulation

```
class Time(object):
    """Class to repr times of day. """

    def getHour (self):
        """Returns: hour attribute"""
        return self._hour

    def setHour(self, h):
        """ Sets hour to h
        Pre: h is an int in 0..23"""
        assert type(h) == int
        assert 0 <= h and h < 24
        self._hour = d
```

Getter

Setter

NO ATTRIBUTES
in class specification

Method specifications
describe the attributes

Setter precondition is same as the **invariant**

Data Encapsulation

```
class Time(object):
    """Class to repr times of day. """

    def getHour (self):
        """Returns: hour attribute"""
        return self._hour

    def setHour(
        """ Sets ho
        Pre: h is a
        assert type(h) == int
        assert 0 <= h and h < 24
        self._hour = d
```

Getter

Setter

NO ATTRIBUTES
in class specification

Method specifications
describe the attributes

Hidden attribute user should **NOT** know about

recondition is
same as the **invariant**

Encapsulation and Specifications

```
class Time(object):
    """Class to represent times of day. """

    ### Hidden attributes
    # Att _hour: hour of the day
    # Inv: _hour is an int in 0..23
    # Att _min: minute of the hour
    # Inv: _min is an int in 0..59
```

No attributes in class spec

These comments make it part of the **class invariant** but not part of the (public) **interface**

These comments do not go in help()

Encapsulation and Specifications

```
class Time(object):
    """Class to represent times of day. """

    ### Hidden attributes
    # Att _h
    # Inv: _h
    # Att _min: minute of the hour
    # Inv: _min is an int in 0..59
```

No attributes in class spec

New style since 2019!

comments part of the **invariant** but not part of the (public) **interface**

These comments do not go in help()

Mutable vs. Immutable Attributes

Mutable

- Can change value directly
  - If class invariant met
  - **Example:** turtle.color
- Has both getters and setters
  - Setters allow you to change
  - Enforce invariants w/ asserts

Immutable

- Can't change value directly
  - May change "behind scenes"
  - **Example:** turtle.x
- Has only a getter
  - No setter means no change
  - Getter allows limited access

May ask you to differentiate on the exam

Easy With Explicit Getters/Setters

```
class Person(object):
    # _name: string or None if unknown (MUTABLE)
    # _born: int > 1900; -1 if unknown (IMMUTABLE)

    def getName(self):
        return self._name

    def setName(self):
        assert value is None or type(value) == str
        self._name = value

    def getBorn(self):
        return self._born
```

Mutable

Immutable

But This Does Not Explain Everything

- Have seen many classes w/o getters/setters
  - `RGB`: Access color values directly
  - `Turtle`: Access positions directly
- How do they enforce invariants?
  - They do have getters/setters!
  - But they are just invisible (???)
- **Will see how in another lesson.**