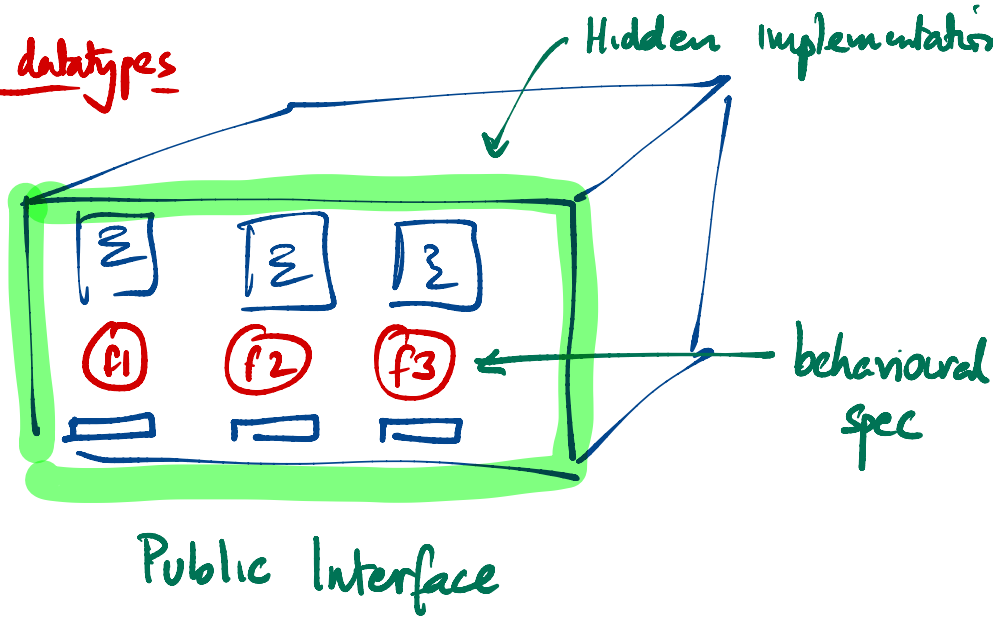
Classes, objects, Java

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 2, 11 January 2024

Abstract datatypes



"Object"

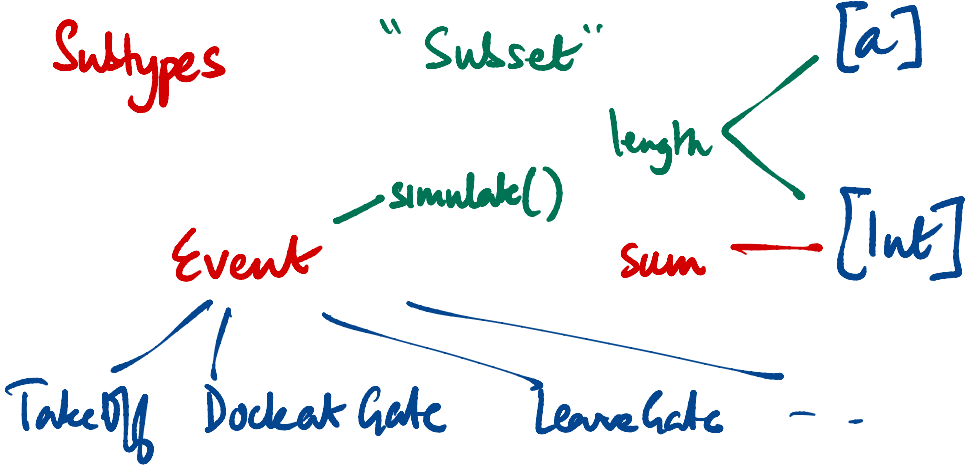
Data Type ⟷ Instances

Class ⟷ Object

Simula

```
loop
    e ← head(eventqueue)
    simulate(e) ⟶ if e is of type A
                  else if e is of type B
                  :
forever
```

Types


Subtypes
"Subset"
[a]
length
simulate()
Event
sum
[Int]
TakeOff
DockatGate
LeaveGate
- -

Heap / Priority Queue — add()
remove_max()

Inheritance

Augmented heap — change_priority()

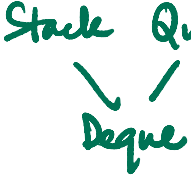
Subtyping — Interfaces compatible
Inheritance — Implementation reuse

deque stack queue

Front Rear

Relationship?

Subtyping

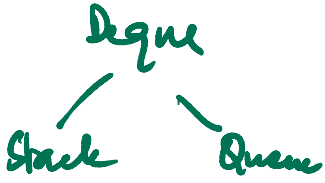


Stack ⇄

Queue ←

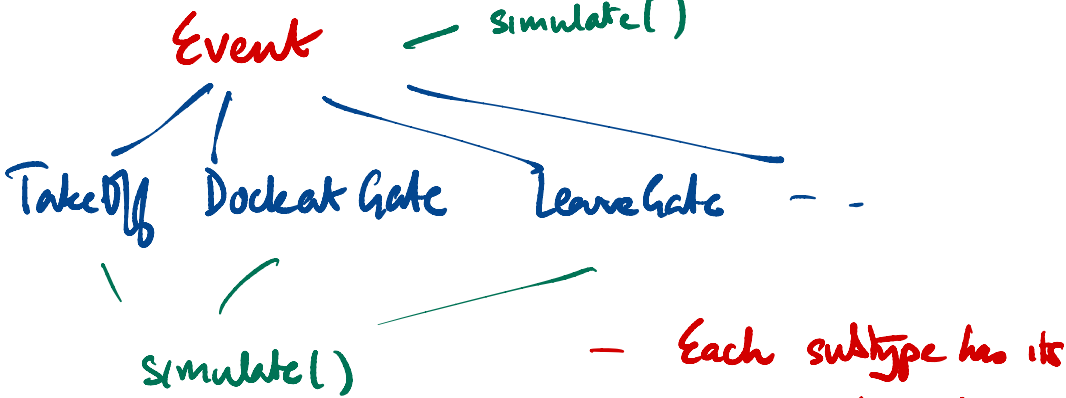
Deque ⇄

Inheritance



←

⇆



"Dynamic despatch"

- Each subtype has its own implement of function
- Called "implicitly"

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template
  - Each object maintains a separate copy of local data
  - Invoke methods on objects — send a message to the object

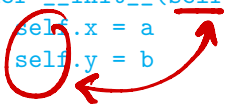
Example: 2D points, in Python

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`

Example: 2D points, in Python

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`
- When we create an object, we need to set it up
  - Implicitly call a constructor function with a fixed name
  - In Python, constructor is called `__init__()`
  - Parameters are used to set up internal values
  - In Python, the first parameter is always `self`

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```



Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy
```

p is a Point

p.translate(7,10)

Point.translate(p,7,10)

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables
- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$
  - Does not update instance variables
  - state of object is unchanged

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy

  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x +
                  self.y*self.y)
    return(d)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)


  def odistance(self):
    return(self.r)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r \cos\theta$, $y = r \sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code
- Rely on programmer discipline

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Rectangle(s,s)

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables
- The following is legal

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Can change the instance variable in `Square`
  - `self.side`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance . . .

- Can change the instance variable in `Square`
  - `self.side`
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`
  - But `s.width` and `s.height` have not been defined!
  - Subtype is not forced to be an extension of the parent type

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` constructor sets `s.width` and `s.height`
  - But the instance variable names have changed!
  - Why should `Square` be affected by this?

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables
- Static type checking catches errors early

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```