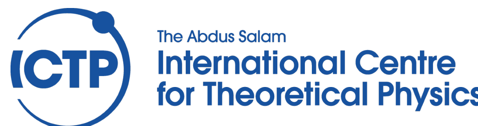
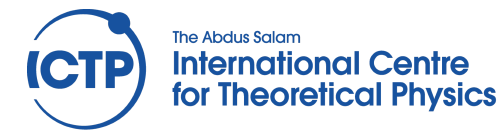
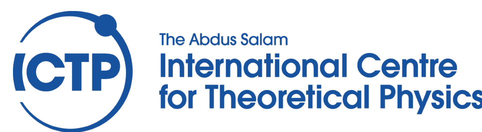


Object-oriented design

David Grellscheid





Programming paradigm examples

Structured / Non-Structured

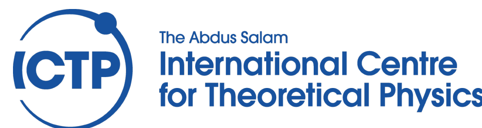
Declarative / Imperative

Procedural

Object-oriented

Functional

(Almost) any style can be implemented in any language

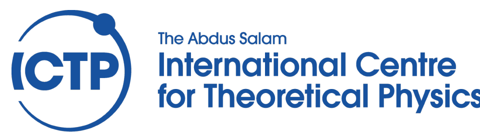


Grady Booch

"Object-oriented analysis and design"

2nd edition

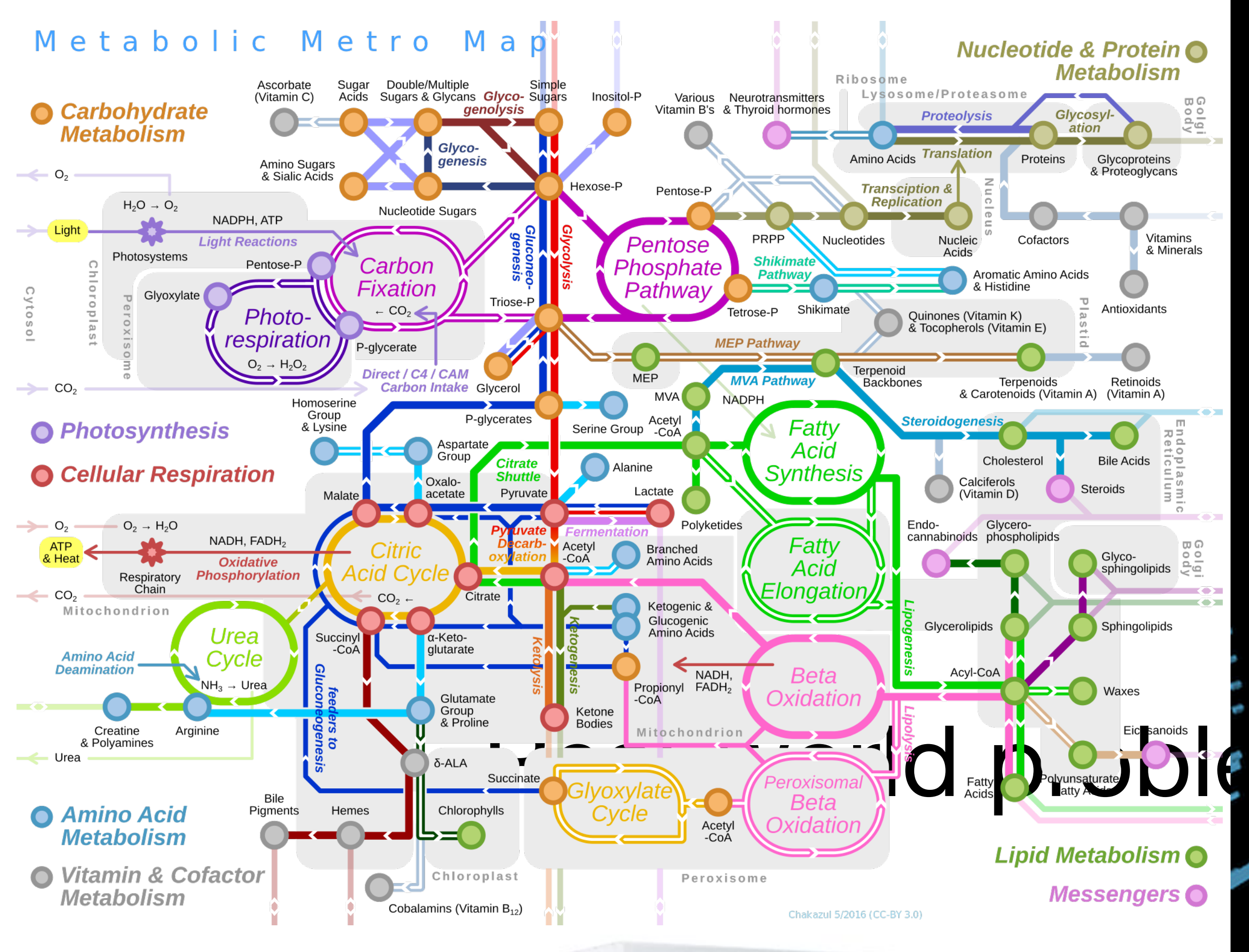
Addison-Wesley, 1994

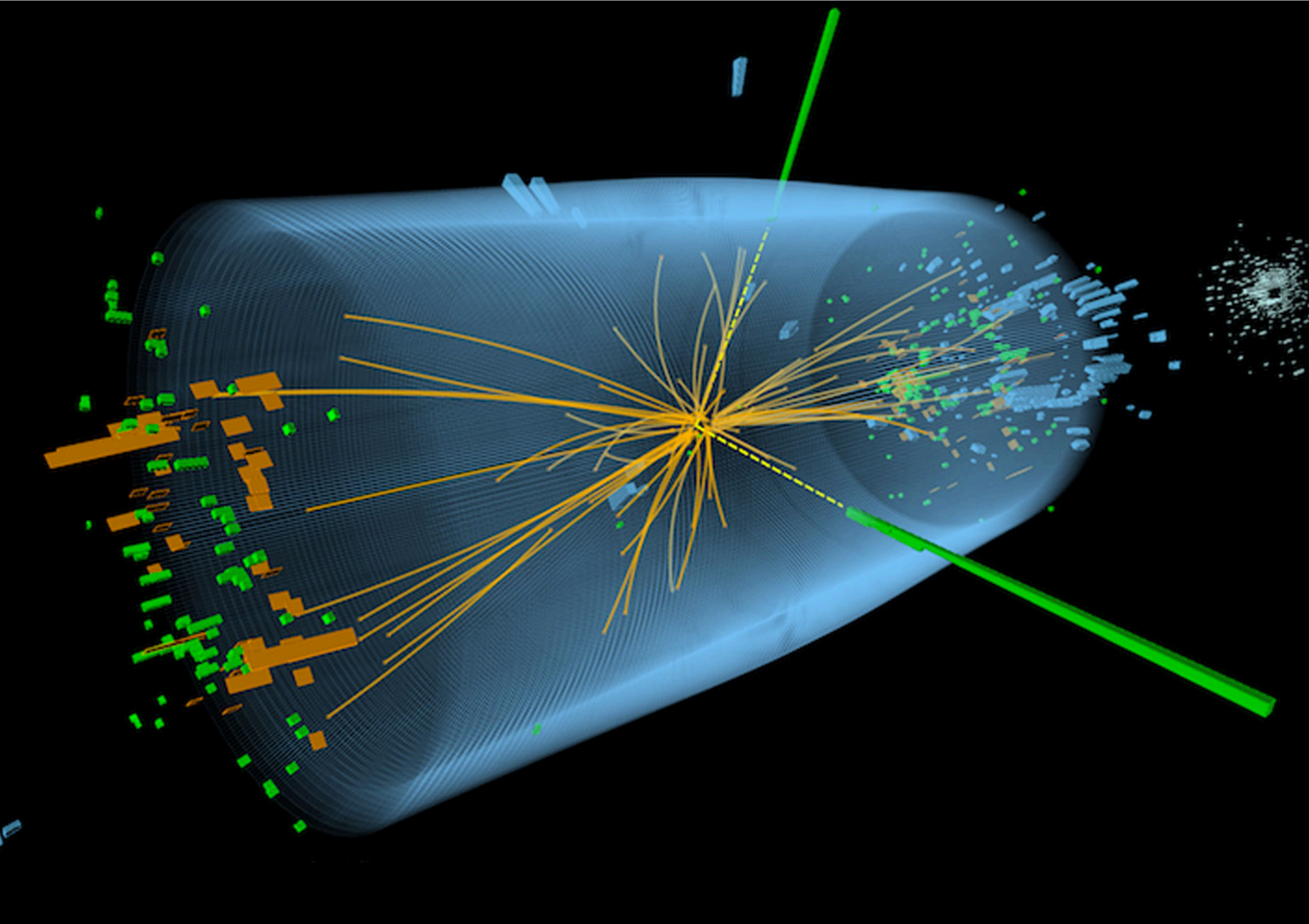


Main goal: manage complexity

Different approaches, OO is just one of them!

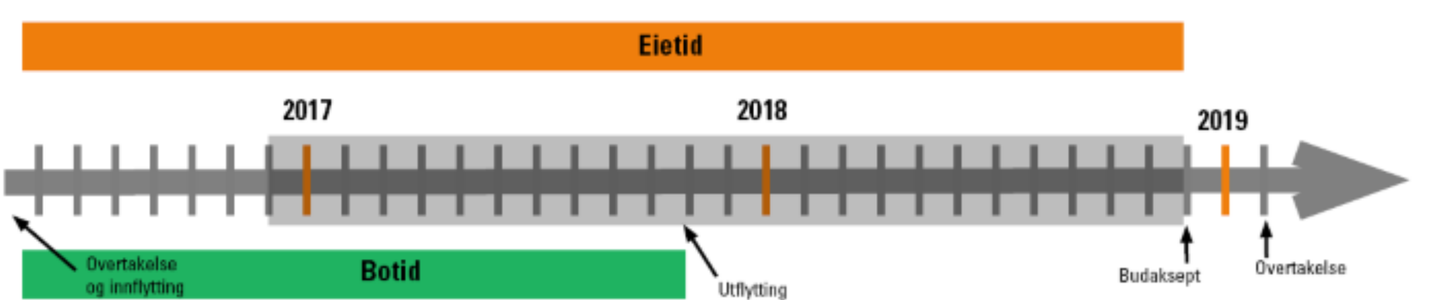
see e.g. Haskell for a completely different approach to complexity handling: functional programming



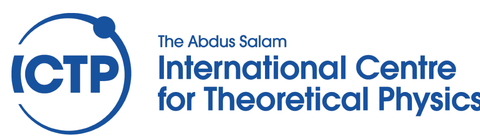


Må du skatte av boligsalget?

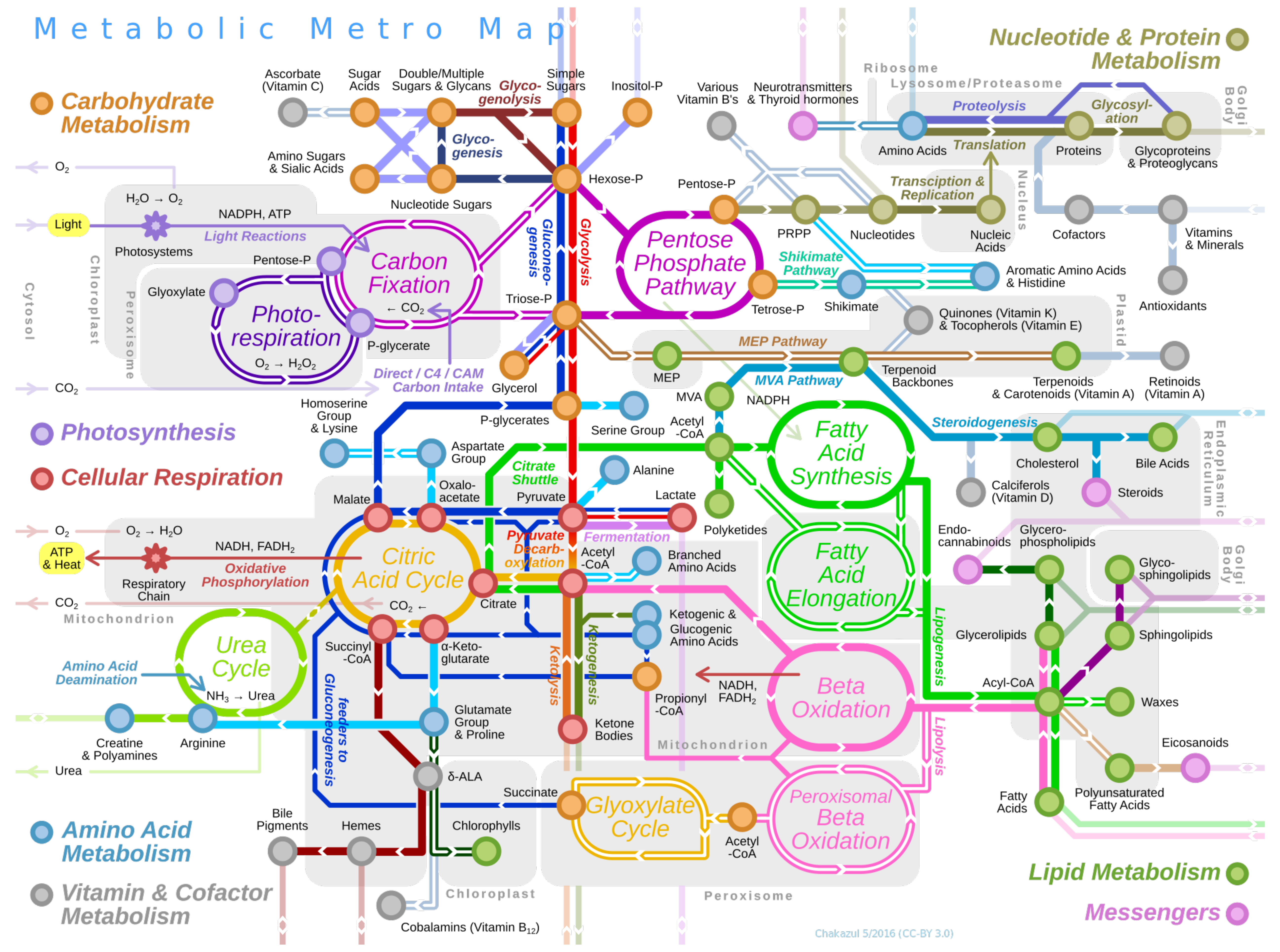
Om du må skatte henger sammen med bo og eietid på boligen du selger. Slik finner du ut dette:

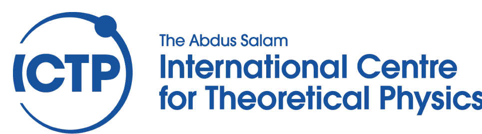


1. Finn datoen for når du aksepterte budet for salg på boligen
2. Tell 24 måneder tilbake fra denne datoen (grått felt i figuren)
3. Har du bodd i boligen i mer enn 12 av disse 24 månedene?
   – Da er salget ikke skattepliktig!
4. Har du ikke bodd i boligen i mer enn 12 av disse 24 månedene?
   – Da er salget mest sannsynlig skattepliktig, og du må skatte av gevinsten.

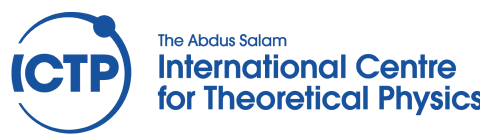


- **Complexity of the problem domain**
  external; requires software maintenance, evolution, preservation
- **Development process**
  impossible for one developer to understand large projects completely
- **Software is boundlessly flexible**
  able to work at any level of abstraction; no fixed quality standards
- **Behaviour of discrete system**
  natural world physics is local and continuous
  program state is not: combinatoric, small change -> large effect


Metabolic Metro Map
Carbohydrate Metabolism
Photosynthesis
Cellular Respiration
Amino Acid Metabolism
Vitamin & Cofactor Metabolism
Nucleotide & Protein Metabolism
Lipid Metabolism
Messengers
Ascorbate (Vitamin C)
Sugar Acids
Double/Multiple Sugars & Glycans
Glycogenolysis
Simple Sugars
Inositol-P
Glycogenesis
Amino Sugars & Sialic Acids
Nucleotide Sugars
Hexose-P
Gluconeogenesis
Glycolysis
Pentose Phosphate Pathway
Pentose-P
Tetrose-P
Triose-P
Glycerol
P-glycerates
Light
Photosystems
Light Reactions
NADPH, ATP
$H_2O \rightarrow O_2$
O_2
CO_2
Carbon Fixation
$\leftarrow CO_2$
Photorespiration
$O_2 \rightarrow H_2O_2$
Glyoxylate
P-glycerate
Direct / C4 / CAM Carbon Intake
Cytosol
Chloroplast
Peroxisome
Various Vitamin B's
Neurotransmitters & Thyroid hormones
Ribosome
Lysosome/Proteasome
Proteolysis
Glycosylation
Amino Acids
Translation
Proteins
Glycoproteins & Proteoglycans
Golgi Body
Transcription & Replication
PRPP
Nucleotides
Nucleic Acids
Nucleus
Cofactors
Vitamins & Minerals
Antioxidants
Shikimate Pathway
Shikimate
Aromatic Amino Acids & Histidine
Quinones (Vitamin K) & Tocopherols (Vitamin E)
Plastid
MEP Pathway
MEP
Terpenoid Backbones
Terpenoids & Carotenoids (Vitamin A)
Retinoids (Vitamin A)
MVA
MVA Pathway
NADPH
Homoserine Group & Lysine
Serine Group
Acetyl-CoA
Fatty Acid Synthesis
Steroidogenesis
Cholesterol
Bile Acids
Calciferols (Vitamin D)
Steroids
Endoplasmic Reticulum
Aspartate Group
Oxaloacetate
Citrate Shuttle
Pyruvate
Alanine
Lactate
Malate
Pyruvate Decarboxylation
Fermentation
Polyketides
Fatty Acid Elongation
Endocannabinoids
Glycerophospholipids
Glycosphingolipids
$O_2 \rightarrow H_2O$
ATP & Heat
NADH, $FADH_2$
Oxidative Phosphorylation
Respiratory Chain
Mitochondrion
Citric Acid Cycle
Branched Amino Acids
Citrate
$CO_2 \leftarrow$
Ketogenic & Glucogenic Amino Acids
Lipogenesis
Glycerolipids
Sphingolipids
Urea Cycle
Amino Acid Deamination
$NH_3 \rightarrow$ Urea
Succinyl-CoA
α-Ketoglutarate
Ketolysis
Ketogenesis
Propionyl-CoA
NADH, $FADH_2$
Beta Oxidation
Acyl-CoA
Waxes
Creatine & Polyamines
Arginine
Urea
Gluconeogenesis feeders to
Glutamate Group & Proline
Ketone Bodies
Lipolysis
Eicosanoids
δ-ALA
Succinate
Glyoxylate Cycle
Acetyl-CoA
Peroxisomal Beta Oxidation
Fatty Acids
Polyunsaturated Fatty Acids
Bile Pigments
Hemes
Chlorophylls
Cobalamins (Vitamin B_{12})
Chakazul 5/2016 (CC-BY 3.0)

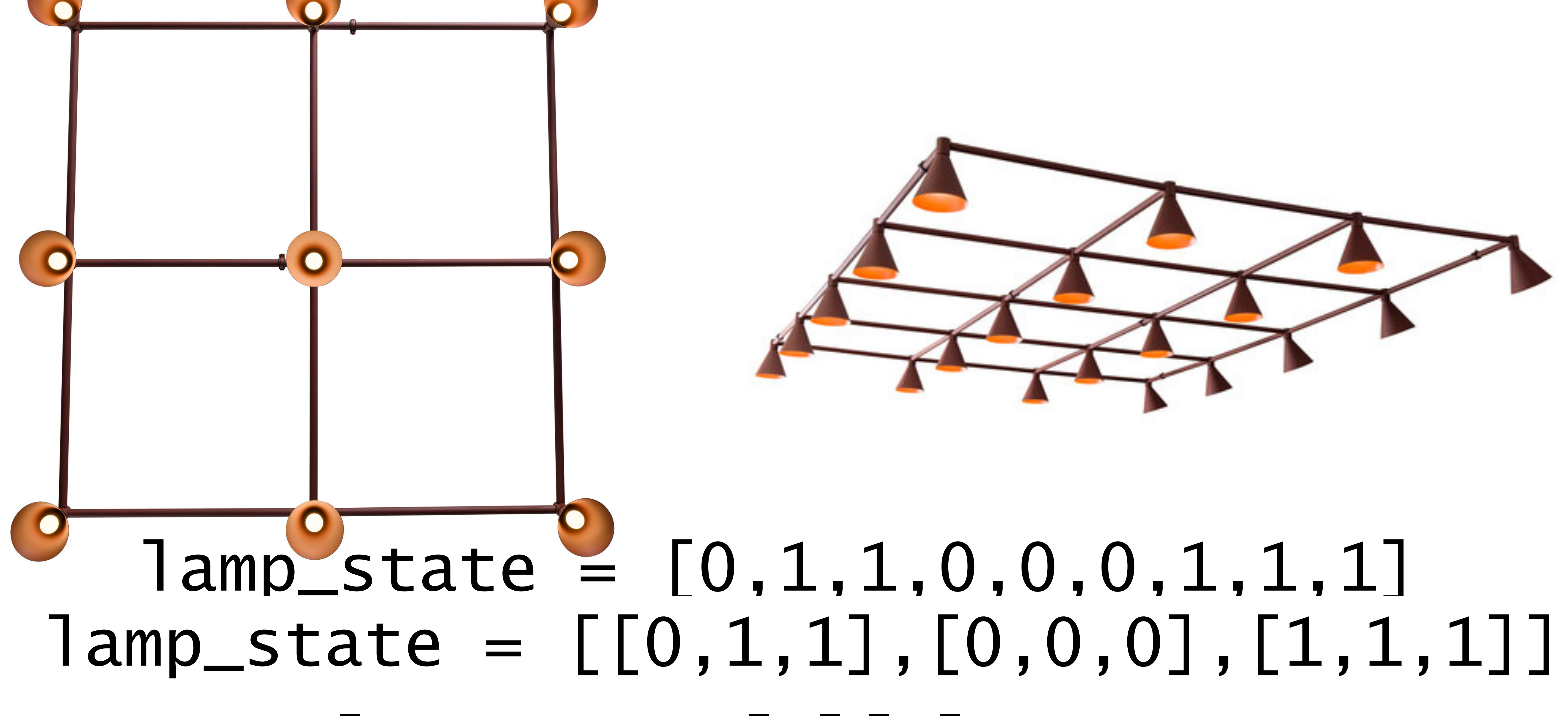


* Complexity is hierarchical grouping of subsystems, down to elementary components
* Choice of elementary blocks is mostly arbitrary
* Links and interactions within a component are much stronger than between components
* Hierarchy uses only a few different subsystems in different combinations
* Working complex systems evolve from working simple systems



- Deal with complexity by decomposition
- Algorithmic decomposition:
  which steps in which order?
- OO decomposition:
  which “real-world” entities are involved?
  how do they relate to each other?

Topic 1: object state



```
lamp_state = [0,1,1,0,0,0,1,1,1]
lamp_state = [[0,1,1],[0,0,0],[1,1,1]]
lamp_state[1][2] == 1
```

Also need two angles for the pointing direction:

```
thetas = [[0.3, 0.4, 0.5],[...],[...]]
  phis = [[0.7, 1.1, 0.0],[...],[...]]
```

Parallel lists are clumsy to use

```
lamp_state = [[  0,   1,   1],[…],[…]]
    thetas = [[0.3, 0.4, 0.5],[…],[…]]
      phis = [[0.7, 1.1, 0.0],[…],[…]]
```

One possible solution: group the other way

```
lamps = [[▯,▯,▯],[▯,▯,▯],[▯,▯,▯]]
```

Now, ▯ represents one lamp

Parallel lists are clumsy to use

3 arrays of values

```
lamp_state = [[  0,   1,   1],[…],[…]]
    thetas = [[0.3, 0.4, 0.5],[…],[…]]
      phis = [[0.7, 1.1, 0.0],[…],[…]]
```

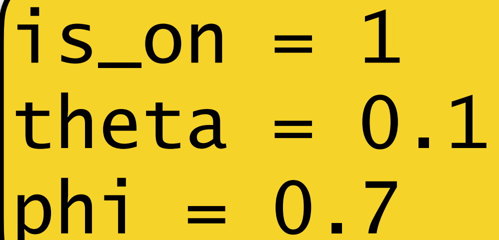
One possible solution: group the other way

```
lamps = [[▯,▯,▯],[▯,▯,▯],[▯,▯,▯]]
```

an array of lamps

Now, ▯ represents one lamp

-is_on
-theta
-phi



is_on = 0
theta = 0.3
phi = 0.1

is_on = 1
theta = 0.2
phi = 0.37

is_on = 1
theta = 0.9
phi = 1.9

...

Class

Lamp

-is_on
-theta
-phi

attributes / member variables

Constructor

is_on = 1
theta = 0.1
phi = 0.7

is_on = 0
theta = 0.3
phi = 0.1

is_on = 1
theta = 0.2
phi = 0.37

is_on = 1
theta = 0.9
phi = 1.9

...

object

object

object

object

```
class Lamp:
    def __init__(self, on=0, th=0, ph=0):
        self.is_on = on
        self.theta = th
        self.phi = ph
```

Constructor

attributes

python

objects

```
l1 = Lamp(1, 0.4, 0.7)
l2 = Lamp(0, 1.1, 0.3)

print(l1.is_on)
print(l2.is_on)

print(l2.theta)
```

```java
public class Lamp {
  int isOn;
  double theta;
  double phi;

  public Lamp(int on, double th, double ph) {
    isOn = on;
    theta = th;
    phi = ph;
  }
}
```

attributes

Constructor

Java

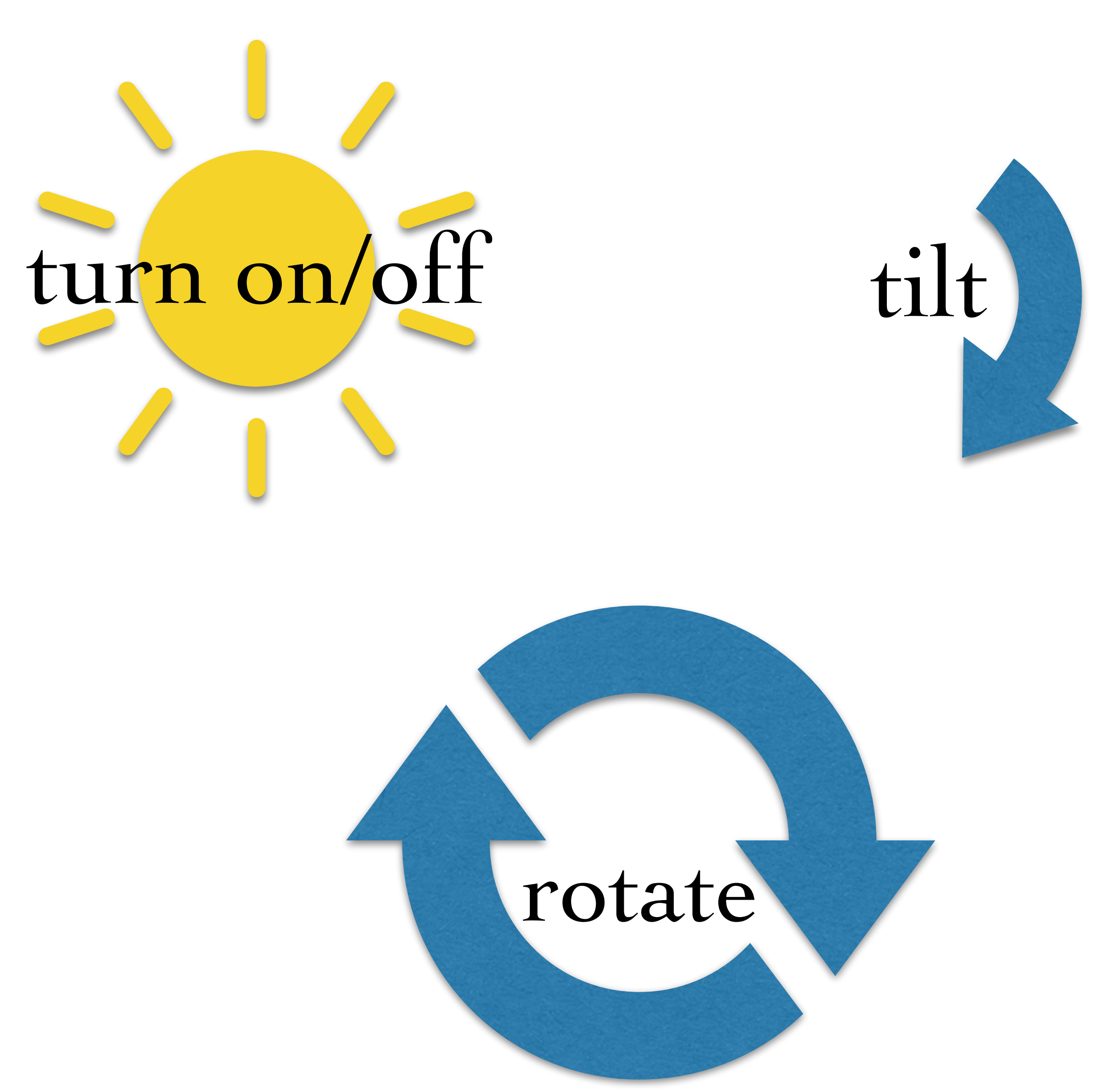
objects

```java
Lamp l1 = new Lamp(1, 0.4, 0.7);
Lamp l2 = new Lamp(0, 1.1, 0.3);

System.out.println(l1.isOn);
System.out.println(l2.theta);
```

Topic 2: object behaviour

We created objects with internal state.
What about lamp behaviours?



```
class Lamp:
    def __init__(self, on=0, th=0, ph=0):
        self.is_on = on
        self.theta = th
        self.phi = ph

    def turn_on(self):
        self.is_on = 1

    def turn_off(self):
        self.is_on = 0

    def rotate(self,angle):
        self.phi += angle
```

Constructor

attributes

methods / member functions

python

```java
public class Lamp {
  int isOn;
  double theta;
  double phi;

  public Lamp(int on, double th, double ph) {
    isOn = on;
    theta = th;
    phi = ph;
  }

 public void turnOn() { isOn = 1; }
 public void turnOff() { isOn = 0; }
 public void rotate(double angle) { phi += angle; }
 public void tilt(double angle) { theta += angle; }

}
```

attributes

Constructor

methods

Java

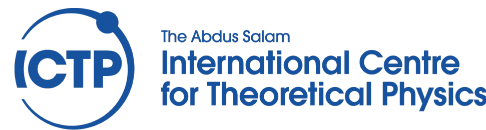
Object methods allow us to use language from the problem domain rather than basic types:

```
lamp_A = Lamp(1, 0.4, 0.7)

lamp_A.turn_off()
lamp_A.rotate(0.2)
```

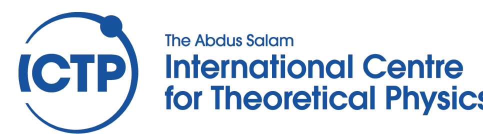
```
Lamp lampA = new Lamp(1, 0.4, 0.7);

lampA.turnOff();
lampA.rotate(0.2);
```



Object

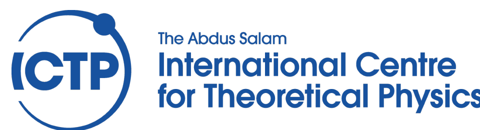
- **State**: inner structure with current values
- **Behaviour**: external interaction and state changes (construct / destruct // modify / select / iterate)
- **Identity:** distinct to all other objects
  It's not the name, one object can have many names!
  Identity considerations are relevant when looking at copying, lifetime and ownership behaviour.



Class

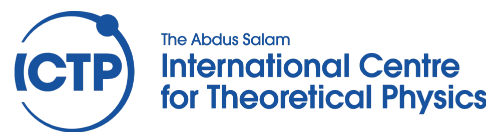
Objects with common structure and behaviour belong to a **class**. The class defines both.

An object is an **instance** of a class.



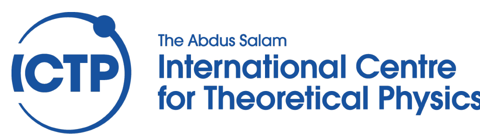
Core features of OO design

* Abstraction
* Encapsulation
* Modularity
* Hierarchy



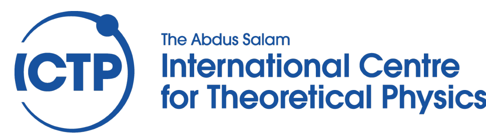
Abstraction

* Outside view of the object
* Focus on relevant details, ignore others
* Define distinction to other objects
* No surprises, no unexpected side behaviour



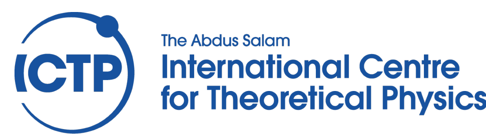
Abstraction

* Identify object invariants, properties that must be true at any time
* Operations have pre- and post-conditions, they must be satisfied
* Objects should never enter inconsistent state



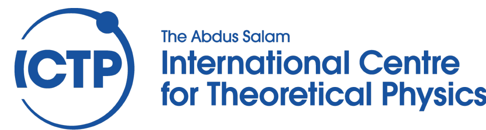
Abstraction

* Implementation details do not matter here
* Define public member functions
* Private section doesn’t matter yet



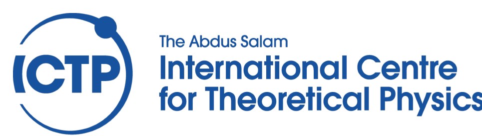
Encapsulation

* separates object's tasks from each other
* actual implementation of the abstraction is hidden
* allows isolated implementation changes
* internal design changes in the objects do not impact the users of the objects



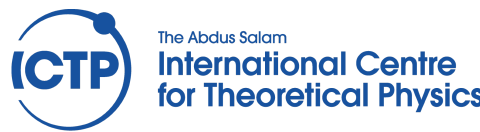
Encapsulation

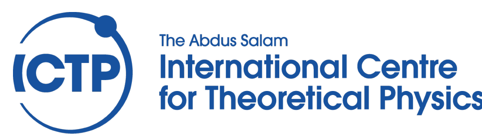
* Abstractions only work well if implementation is encapsulated!



Modularity

- Grouping of classes into functionally related units. Modules should be loosely coupled externally.
- “Physical” collection of units in files, rather than abstract connections
- Difficult to get right first time, may need several redesigns during development


The Abdus Salam
International Centre
for Theoretical Physics

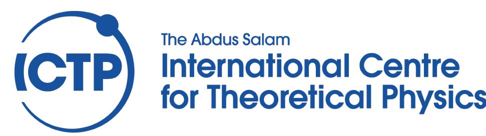


Hierarchy

* Abstractions form hierarchies
* Helps to think about the useful levels

Two main kinds:

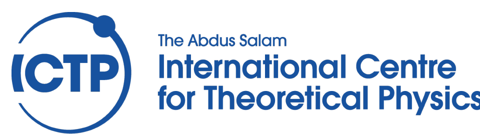
* “is-a”: cat is an animal; oak is a plant
* “has-a”: car has an engine; house has a door



Hierarchy: “is-a”

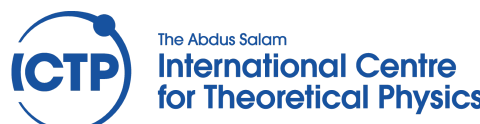
* Modelled by inheritance
* Common functionality moves to the top; applies to all classes down the hierarchy

Easy re-use of code alone is **not** a good reason for inheritance

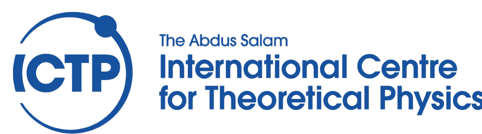


Hierarchy: “has-a”

- Modelled by aggregation
- Objects have other objects as member variables



… main message …

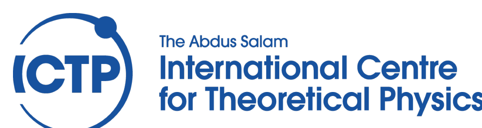


Hierarchy

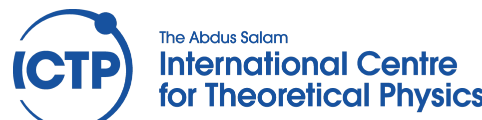
- Abstractions form hierarchies
- Helps to think about the useful levels

Two main kinds:

- “is-a”: cat is an animal; oak is a plant
- “has-a”: car has an engine; house has a door



Exercise



Exercise: a freight station

City A

City B

City C

City D

City E

City A

City B

max 200t D

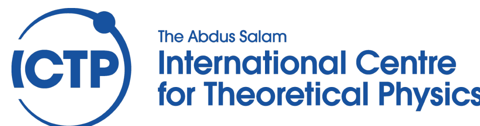
max 450t B

etc…

City C

City D

City E



City A

City B

City C

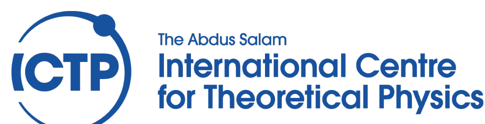
City D

City E

E 1kg frf

C 3kg bfcad

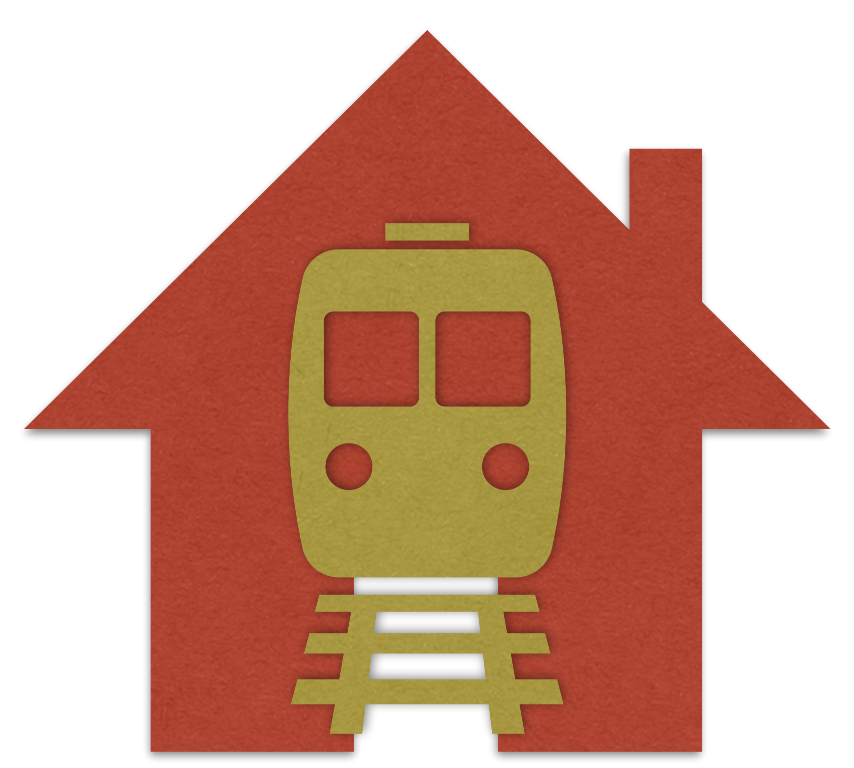
A 5kg cwzvx



Design an OO model for the station

classes, objects, interfaces, public/private, which methods/state

but no implementation!



a random train arrives, is loaded with correct mail, leaves, and repeat

max 200t D