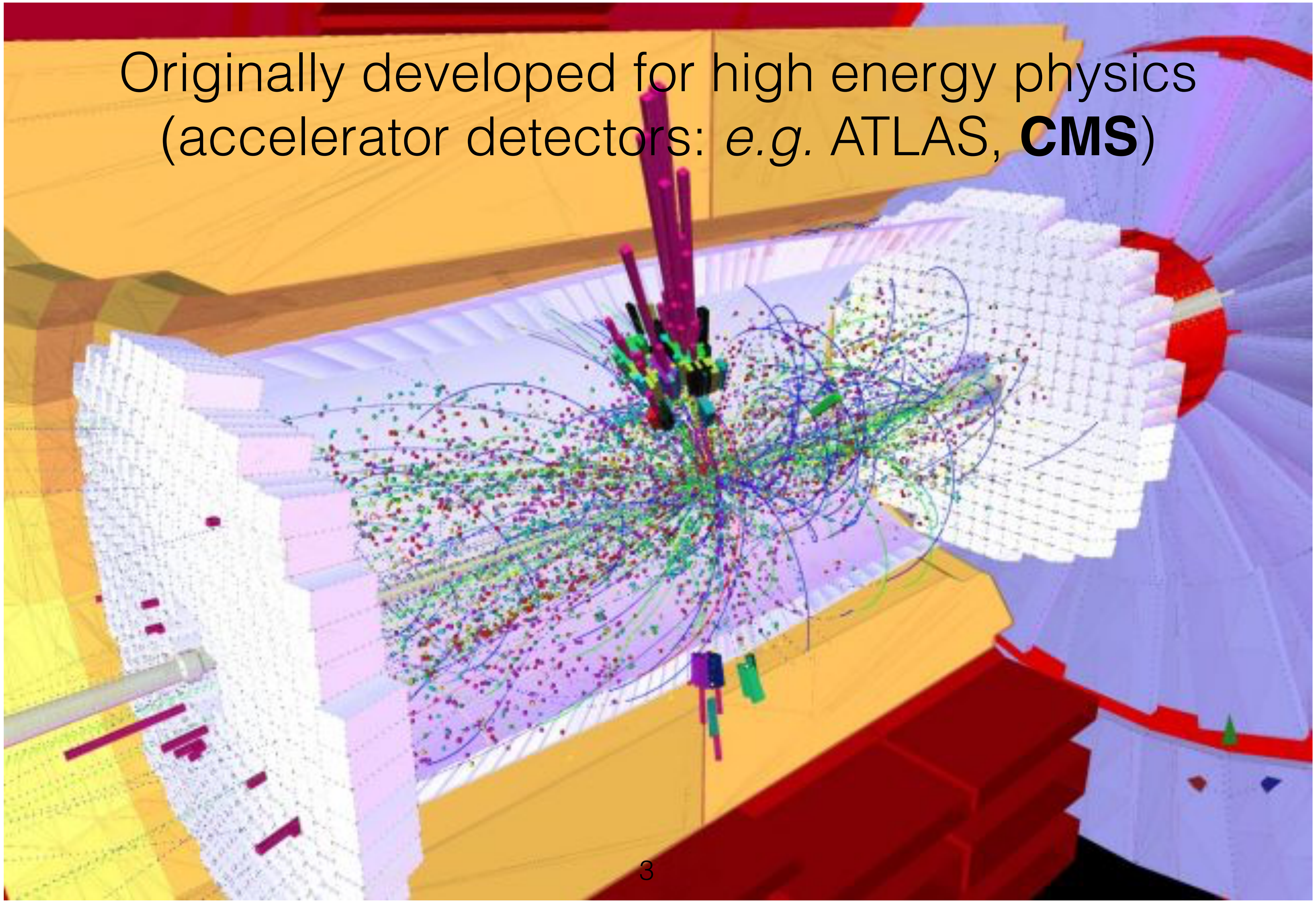
# Lesson 2
# Introduction to GEANT4

## GEANT4

- GEANT = **GE**ometry **AN**d **T**racking
- Software framework for Monte Carlo simulation
  - We don't need to generate and sample distributions, someone already did it for us!
- Specifically for simulation of **particle interactions with matter**
- Following Object Oriented Programming (OOP) paradigm (C++)
- Open source — we can see (and modify) the code!
- Development started at CERN in the late 90's / early 00's for the LHC experiments, now spread by several institutes around the world
- Based on GEANT-3 (started in the 70's, written in Fortran)

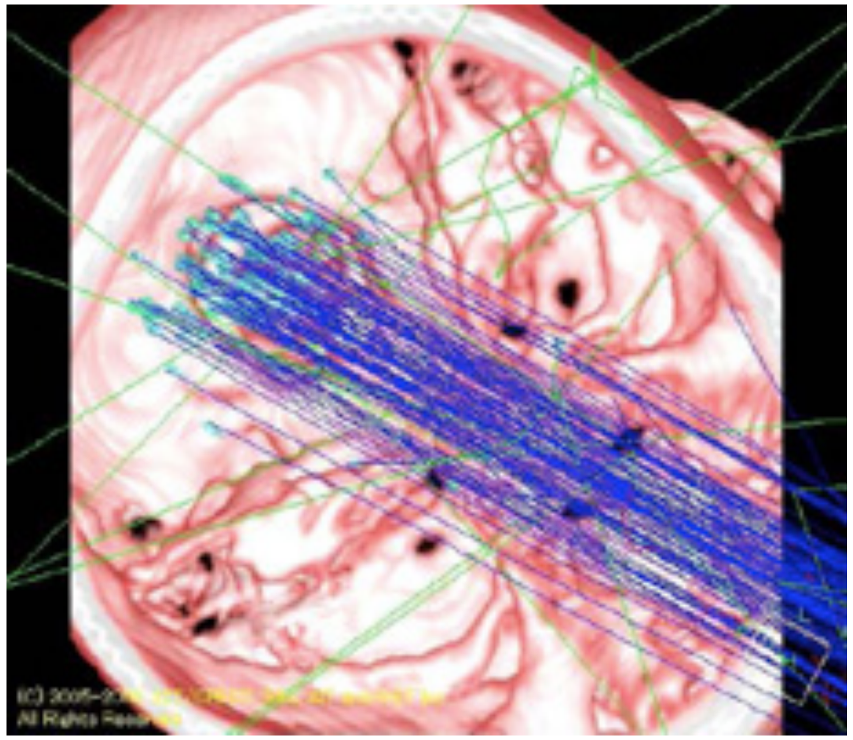
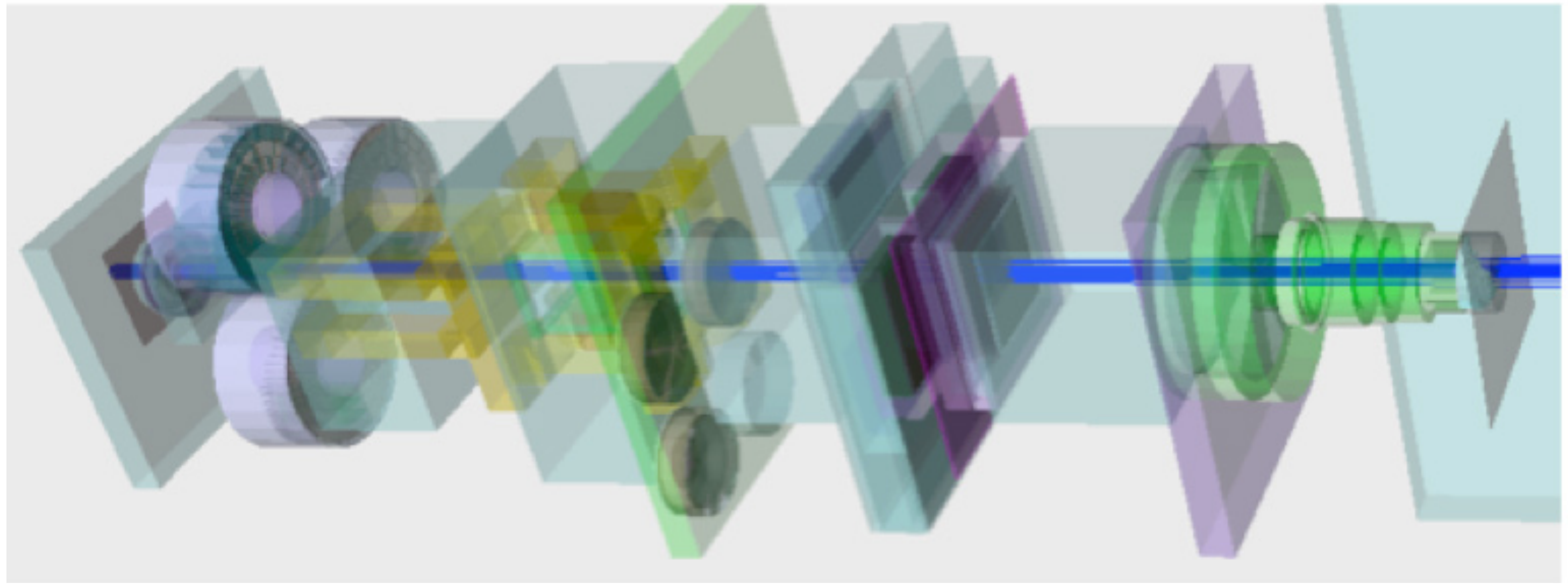
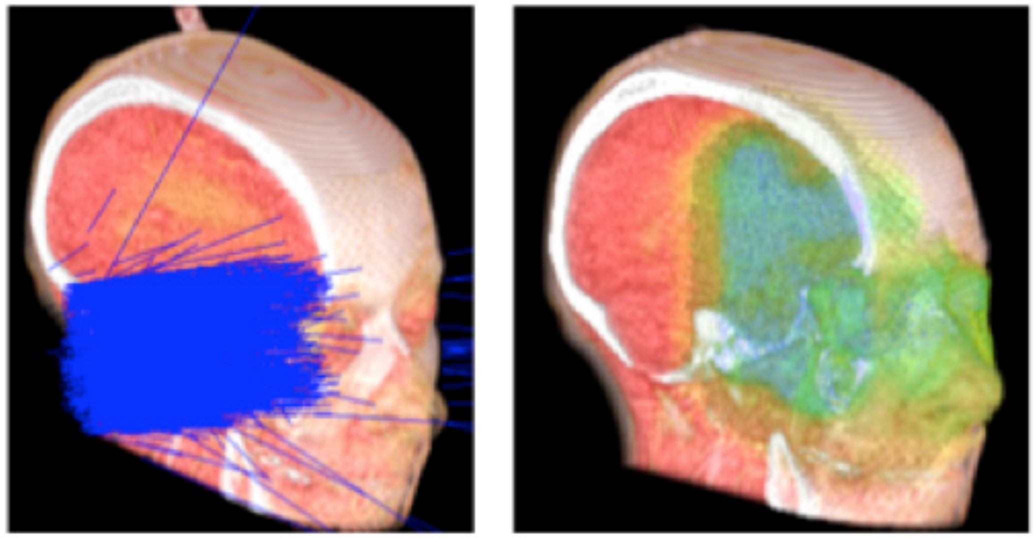
# GEANT4 use cases



Originally developed for high energy physics (accelerator detectors: *e.g.* ATLAS, **CMS**)

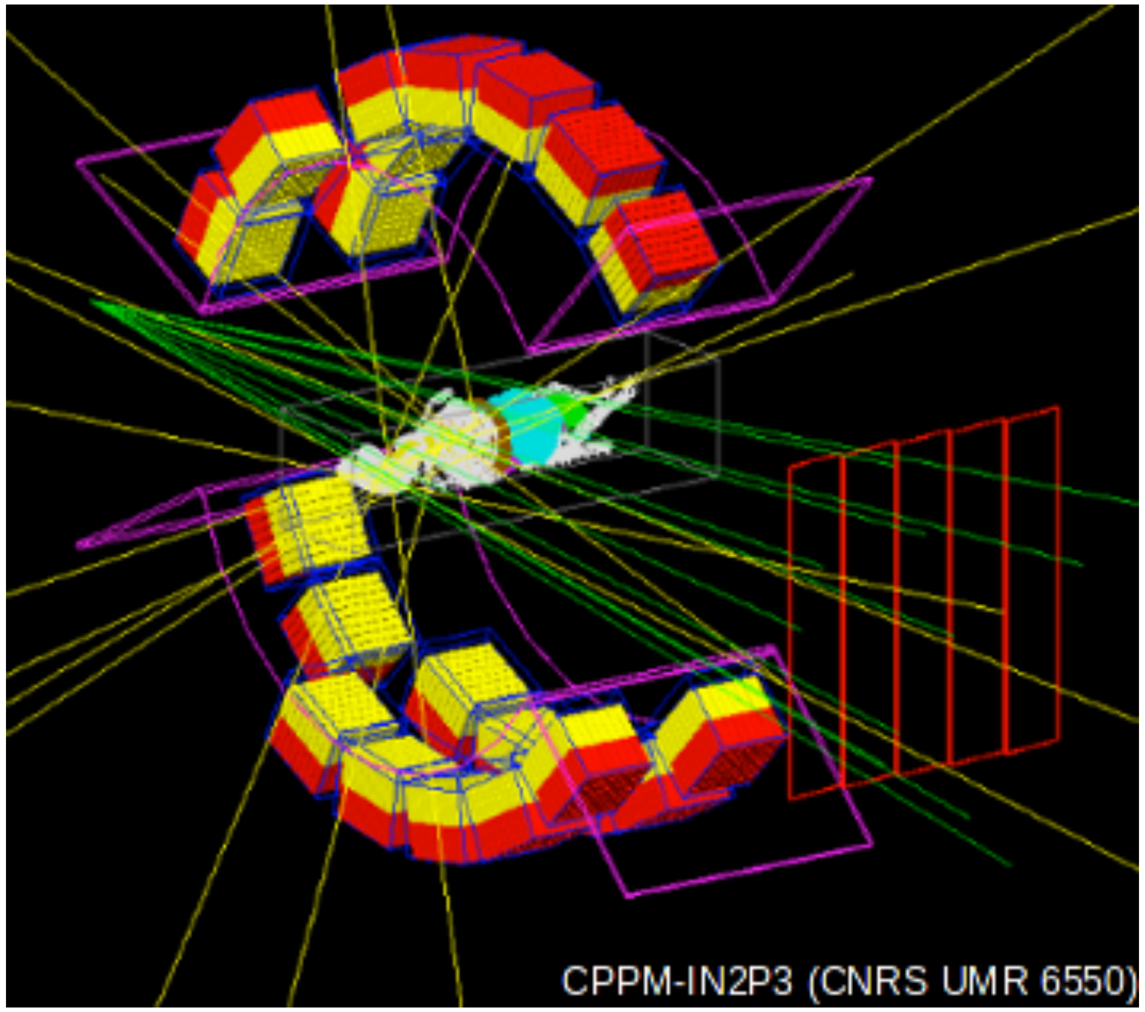
# GEANT4 use cases

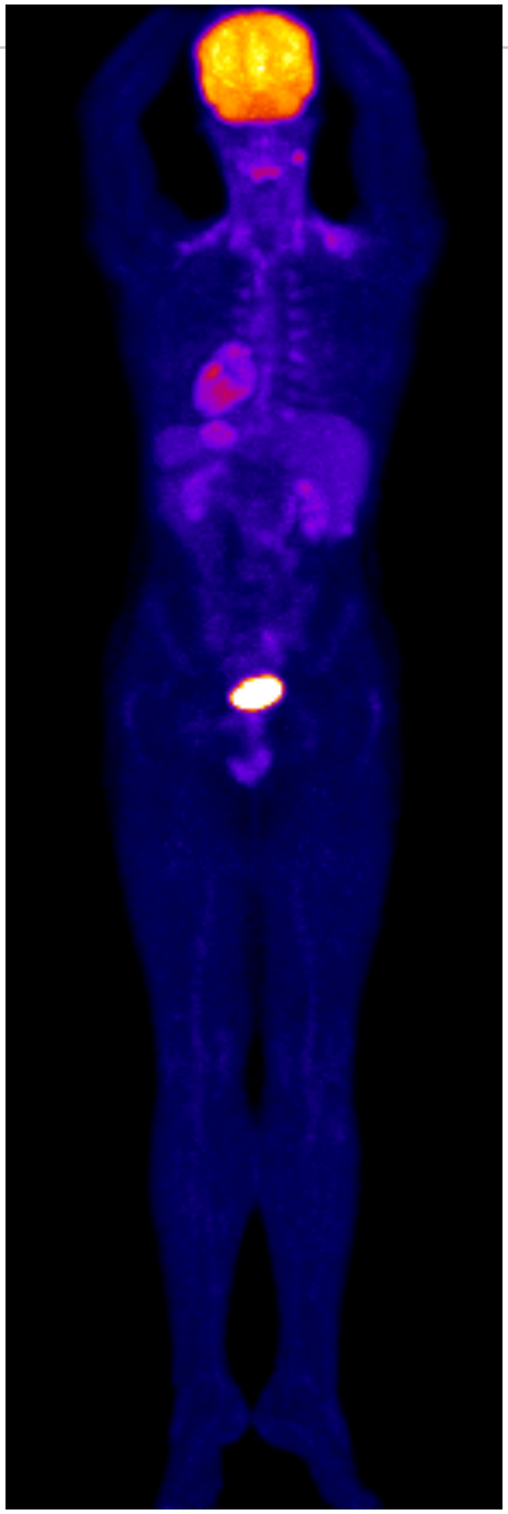
- **Medical Physics**
  - Develop PET and SPECT systems
  - Plan radiation therapy
  - Dosimetry estimates

# GEANT4 use cases

- Dedicated applications (*e.g.* **GATE**, TOPAS)
  - Using macros (instead of coding)
  - Commonly used geometry elements
  - Pre-built physics lists
  - Standard generators for primary particles



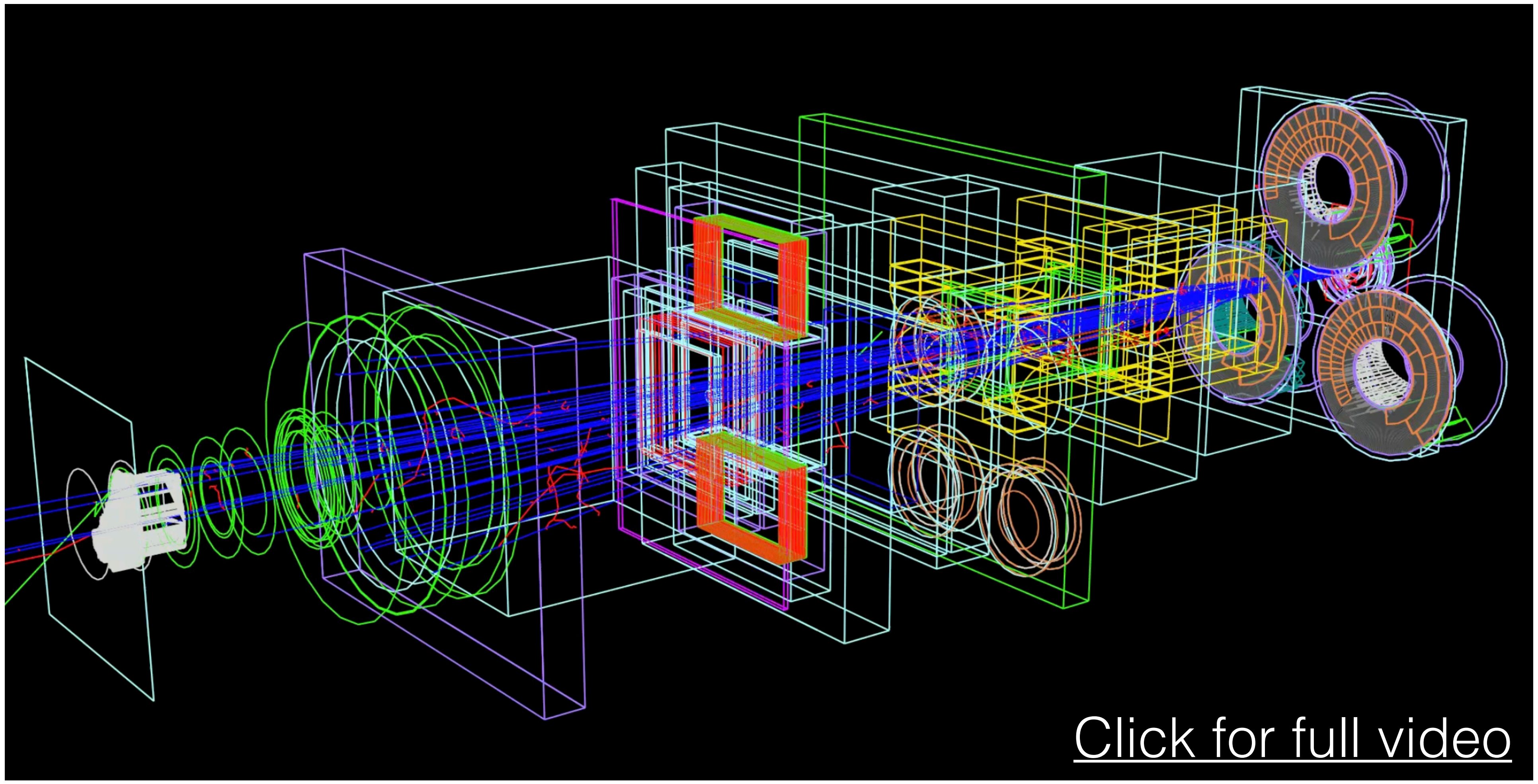


GATE - http://www.opengatecollaboration.org

# GEANT4 use cases

- Dedicated applications (*e.g.* GATE, **TOPAS**)
  - Using macros (instead of coding)
  - Commonly used geometry elements
  - Pre-built physics lists
  - Standard generators for primary particles

TOPAS - https://www.topasmc.org


Click for full video

# GEANT4 use cases

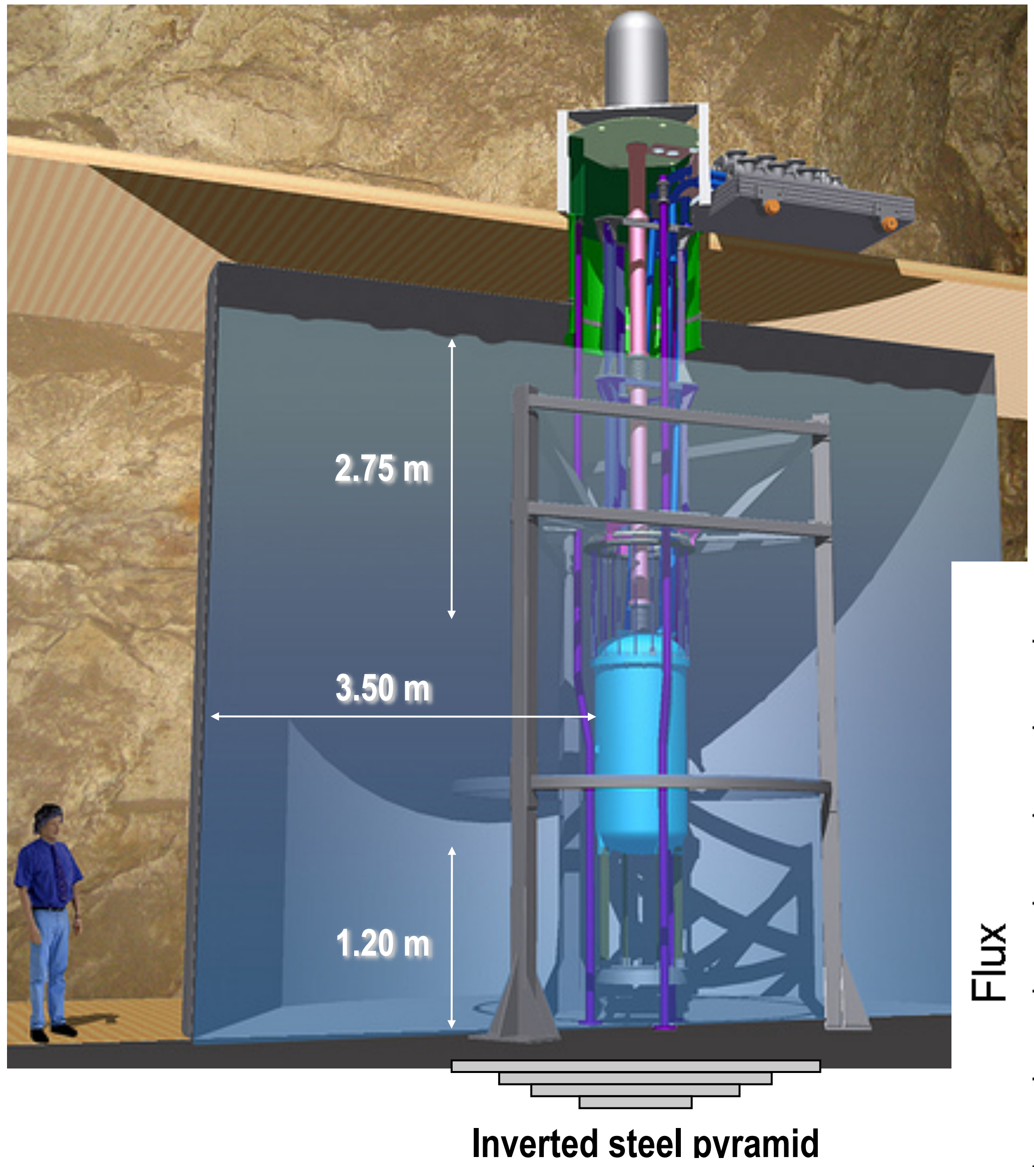
## Space instrumentation

- Study the effects of high-energy radiation in materials and living tissues
- Study the response of the detectors

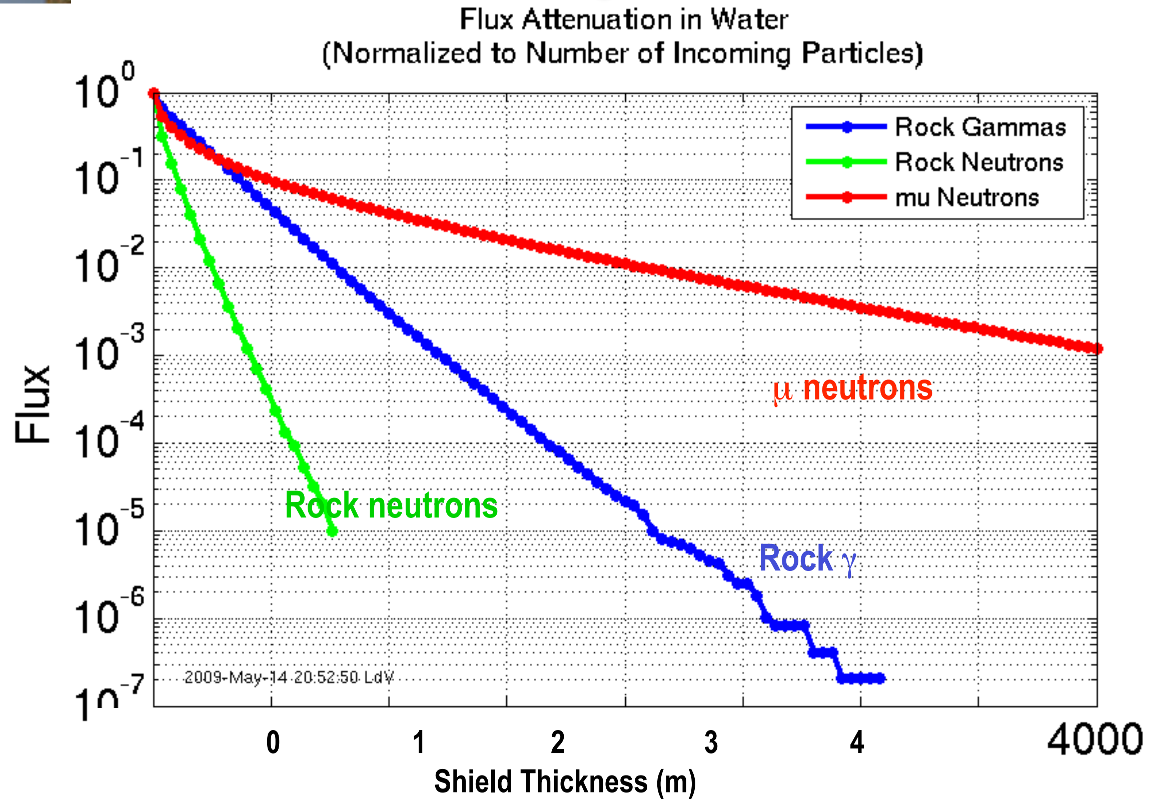
ESA's XMM-Newton telescope

# GEANT4 use cases

- **Underground Physics** (dark matter, neutrinos, 0$\nu\beta\beta$)



Study effectiveness of the water shielding for environment radiation



# GEANT4 Philosophy

- GEANT4 is **a library of tools** for Monte Carlo simulation (in the form of C++ classes)
  - **The user must build his/her own application**
    - this means writing a main C++ program and include GEANT4 classes
    - some software tools built on top of GEANT4 provide several typical and tuneable use cases, especially in medical physics — *e.g.* GATE, TOPAS
  - In order to do that, we need to:
    - ✓ Build the geometry of our experiment (materials, volumes, positions)
    - ✓ Define how each event starts (primary particles)
    - ✓ Choose the physics to use
      - ➡ not like the real world — we can turn off physical processes!
    - ✓ Extract useful information from the simulation, for further analysis

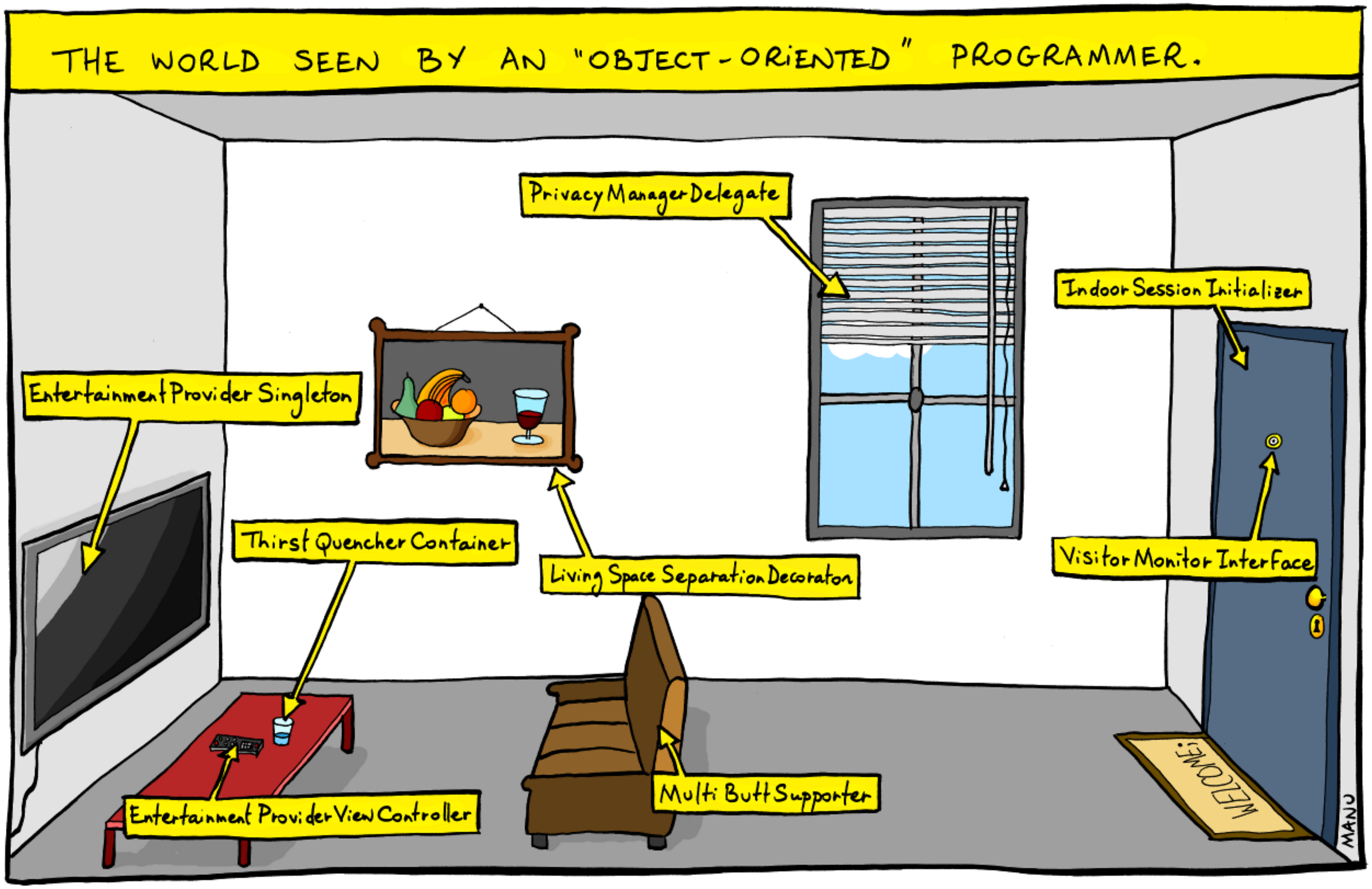
The Application Developer's Guide can be found here

# GEANT4 Philosophy

- GEANT4 provides
  - ✓ Different types of physical processes
    - ✴ Electromagnetic, hadronic, decay, optical
    - ✴ Often more than one model available for the same process
      (sometimes the problem is actually to decide which model to use)
  - ✓ Common particles and their properties
  - ✓ Geometrical solids to build our detectors
  - ✓ A navigator that tracks (follows) each particle as it propagates in the detector (includes support for electric and magnetic fields)
  - ✓ Visualisation (geometry, tracks, hits)
  - ✓ Analysis tools (for online analysis and exporting results)

# Object Oriented Programming

in five slides…



THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
PrivacyManagerDelegate
IndoorSessionInitializer
EntertainmentProviderSingleton
ThirstQuencherContainer
LivingSpaceSeparationDecorator
VisitorMonitorInterface
MultiButtSupporter
EntertainmentProviderViewController
WELCOME!
MANU

# Object Oriented Programming

- You're probably used to program following a "procedural" style
  - ➡ A main code section, with multiple auxiliar functions
  - ➡ This is typical in some languages, such as C, Fortran, Python
  - ➡ OOP is not an exclusive of C++ (Java, Python, Delphi, etc.)
    - ➡ And you can still write "procedural" code in these languages

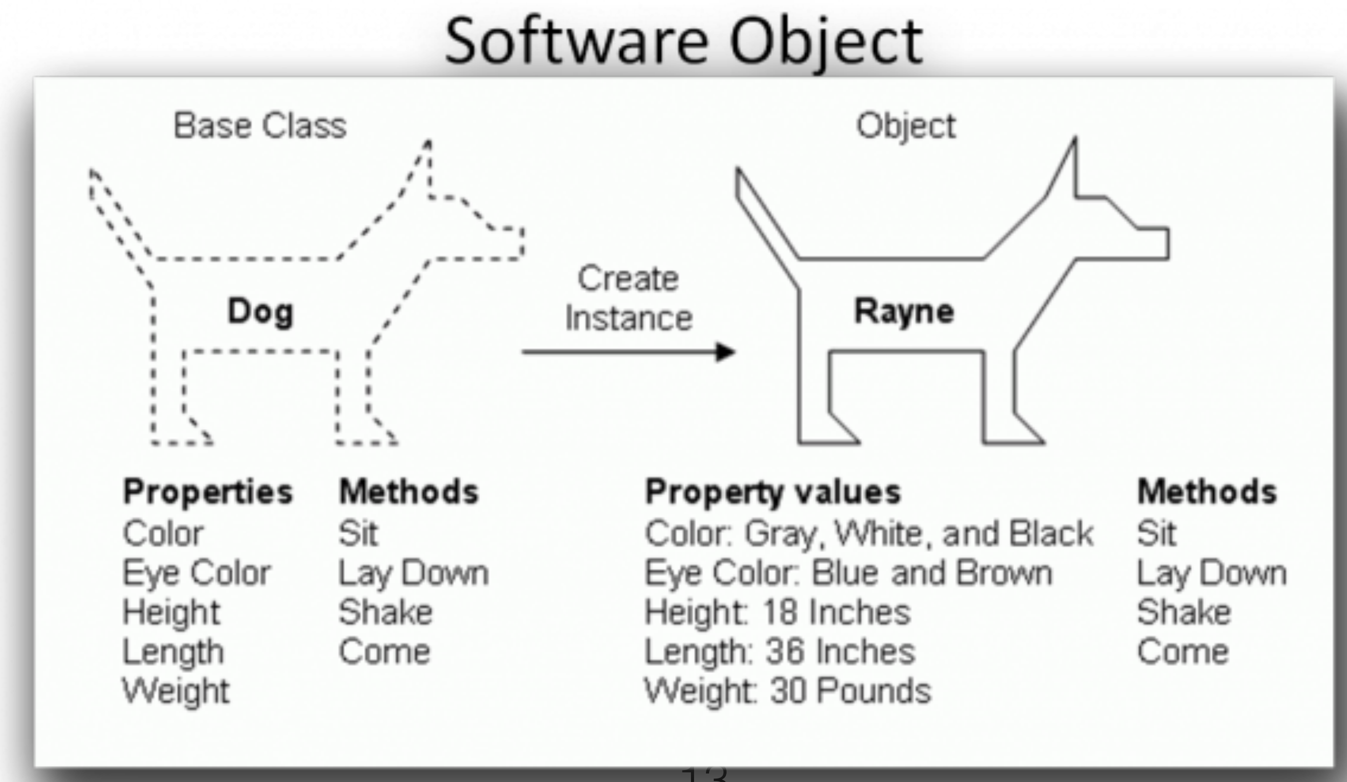
- OOP paradigm: all the elements that make up a program are "objects". They have properties and we can interact with them (send and receive information/actions)

- In a OOP-style simulation, having "independent" objects makes things more natural and closer to the real world
    - ➡ particles, materials (and also isotopes, elements), geometrical solids, volumes
    - ➡ physical processes/models
    - ➡ tracks, events and runs
    - ➡ …

## Object Oriented Programming

- Definition of “object”:
    - An instance — compiled, running version — of a **class**
    - A class is an independent and self-contained block of code
    - You can think of it as a program that can run independently and is waiting to interact with the “outside world” — *i.e.* other objects
    - It may not do anything until ordered to by another program or object

The base class is generic

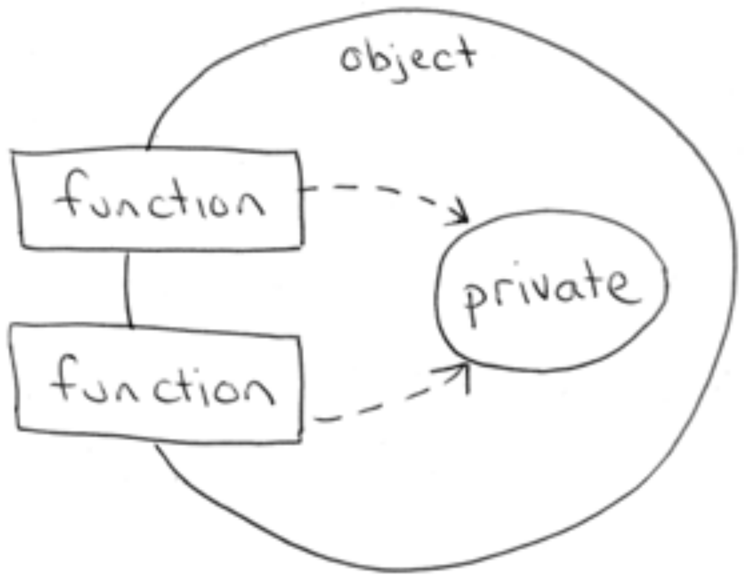


The object is specific

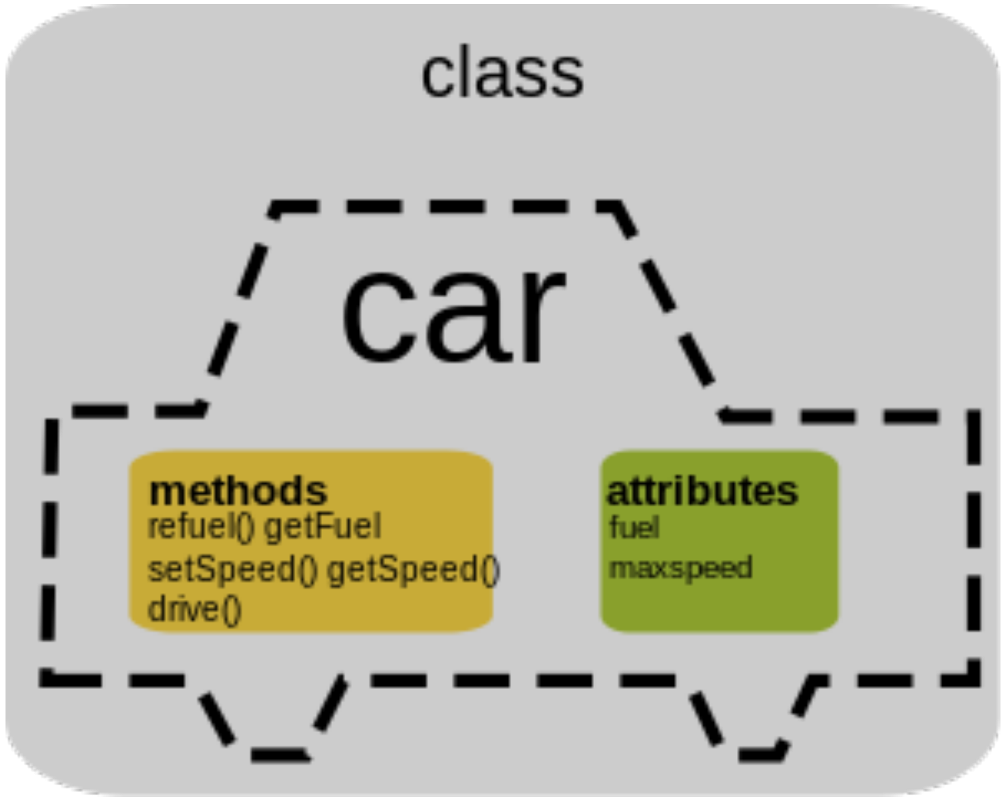
We can have multiple objects of the same base class running at the same time

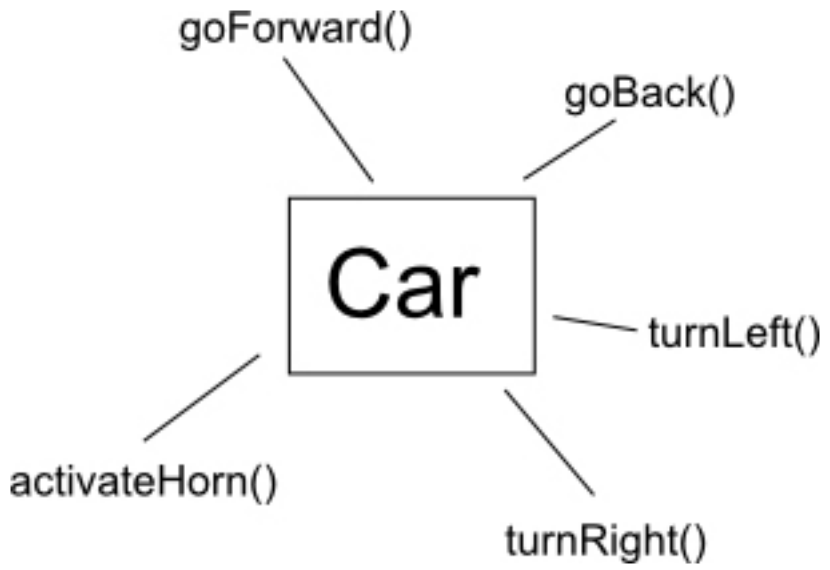
## Object Oriented Programming

- Encapsulation:
  - ➡ A class has functions (usually called methods) to interact with the "exterior world" (other objects or programs) or perform internal tasks
  - ➡ Usually has several internal (private) variables, which can only be accessed or modified using the methods of the class

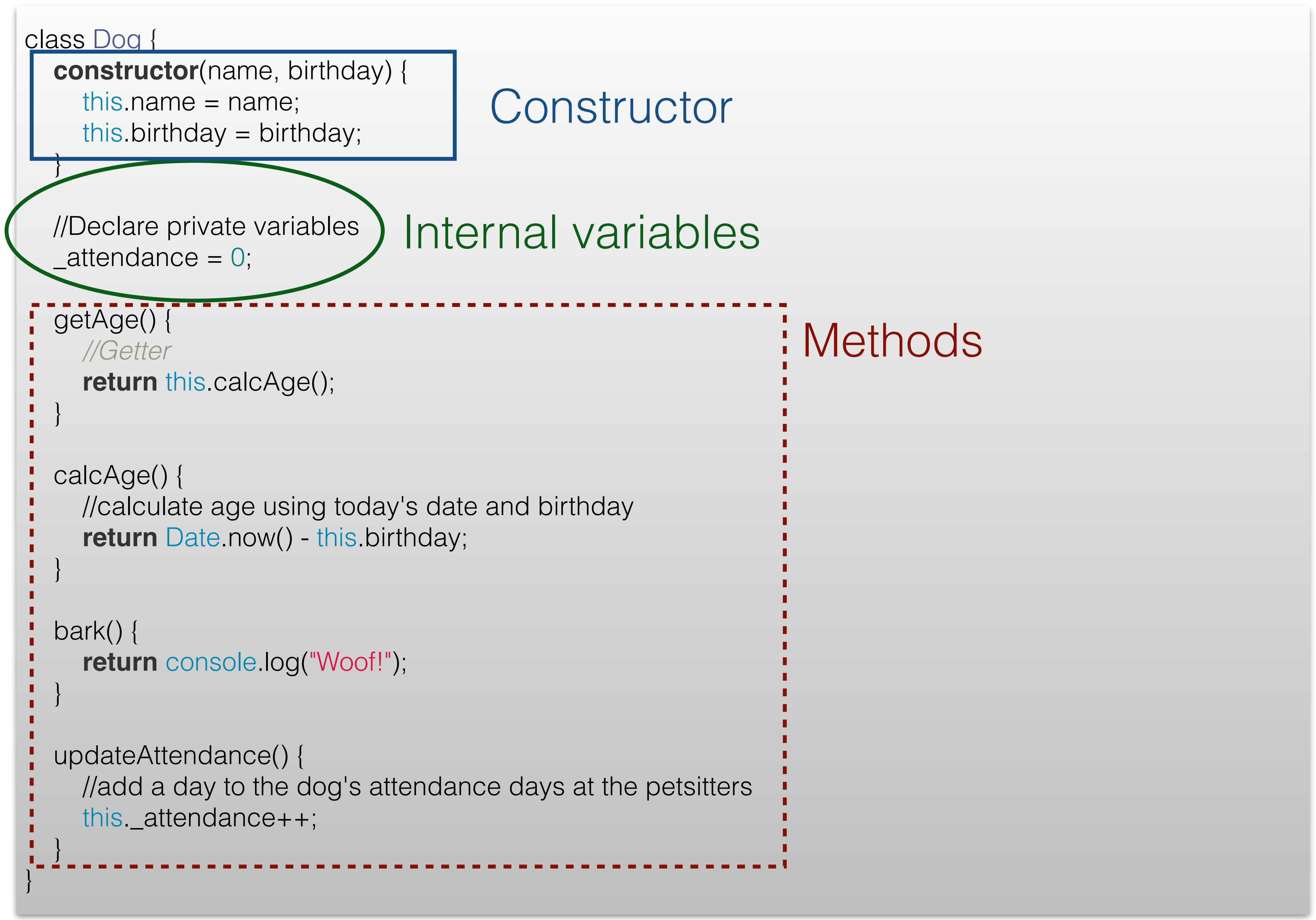


Encapsulation protects class variables from direct access by users





# Object Oriented Programming



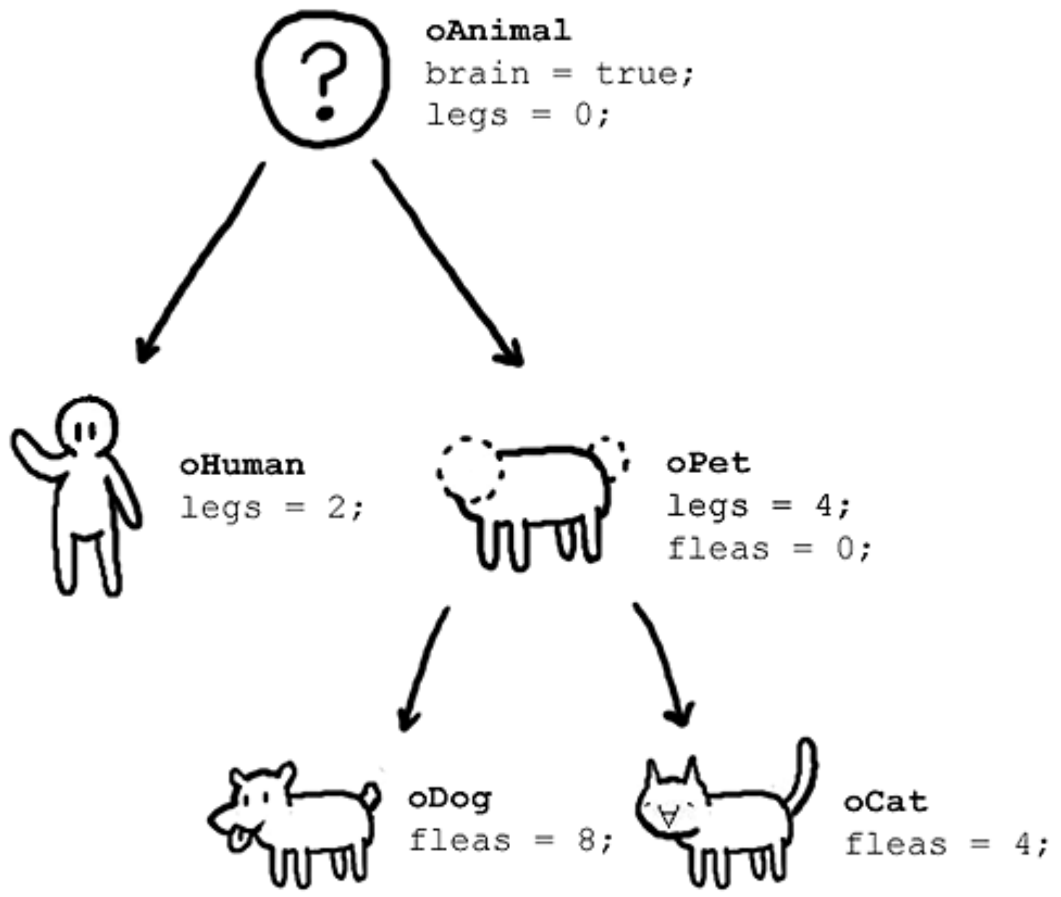
```
class Dog {
  constructor(name, birthday) {
    this.name = name;
    this.birthday = birthday;
  }

  //Declare private variables
  _attendance = 0;

  getAge() {
    //Getter
    return this.calcAge();
  }

  calcAge() {
    //calculate age using today's date and birthday
    return Date.now() - this.birthday;
  }

  bark() {
    return console.log("Woof!");
  }

  updateAttendance() {
    //add a day to the dog's attendance days at the petsitters
    this._attendance++;
  }
}
```

```
//instantiate a new object of the Dog class, and individual dog named Rufus
const rufus = new Dog("Rufus", “2/1/2017");
rufus->getAge();
rufus->bark();
```

Code samples from here

# Object Oriented Programming

- Inheritance:
    - Classes have inheritance: a class may be “daughter” of another, inheriting its internal variables and methods — and then have additional ones
    - Given their modularity, it’s easy to reuse classes in different applications



Don’t worry if you’re feeling a bit lost. This will become more natural when we start with examples.

# A GEANT4 example

## Base class

**G4ParticleDefinition**

**Methods to access these properties:**

GetParticleName
GetPDGMass
GetPDGCharge
GetPDGLifeTime
etc.

**Particle properties:**

name
mass
charge
life time
etc.

**Methods to modify these properties:**

SetParticleName
SetPDGMass
SetPDGCharge
etc.

## Daughter classes

G4Electron
G4Positron
G4Gamma
G4Alpha
G4Ion
…

## Objects

```
electron1 = new G4Electron();
electron2 = new G4Electron();
e+ = new G4Positron();
gamma = new G4Gamma()
…
```

## Structure of a GEANT4 simulation

- To create a simulation, the user must define (minimum):
    - ➡ ***main()*** - main program — declare classes, initialise managers
    - ➡ ***DetectorConstruction()*** - geometry definitions (materials, volumes)
    - ➡ ***PrimaryGenerator()*** - define initial particles (*primaries*)
    - ➡ ***PhysicsList()*** - particles to use, associated physics processes and models

# Structure of a GEANT4 simulation

- To create a simulation, the user must define (minimum):
  - ➡ ***main()*** - main program — declare classes, initialise managers
  - ➡ ***DetectorConstruction()*** - geometry definitions (materials, volumes)
  - ➡ ***PrimaryGenerator()*** - define initial particles (*primaries*)
  - ➡ ***PhysicsList()*** - particles to use, associated processes and models

**Mandatory classes**

**During these lessons, we will always use pre-made simulation structures and modify them.**

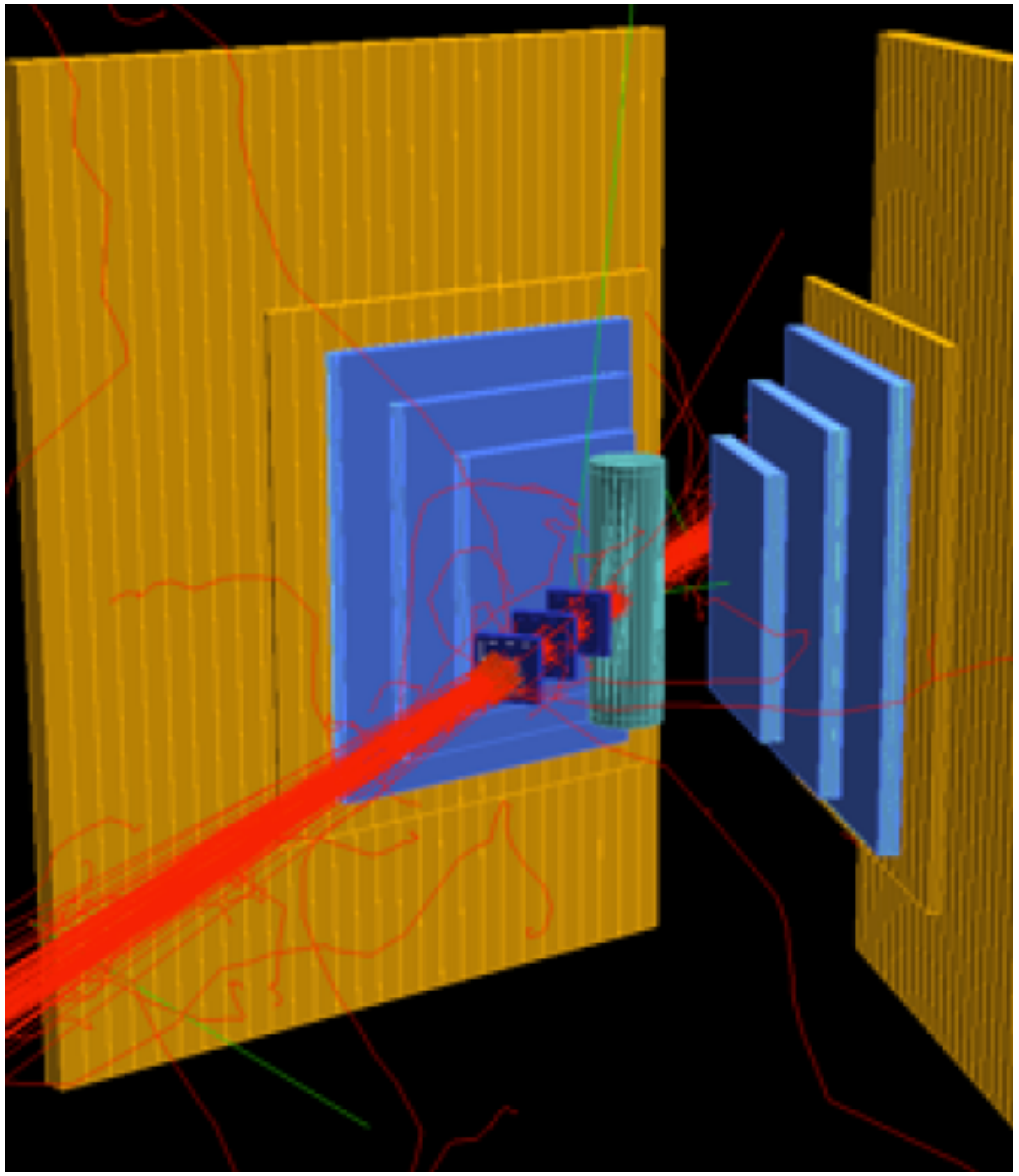
**You will not need to create them from scratch!**

# Structure of a GEANT4 simulation

- Let’s have a look at a simple example…
  - Go to https://lip.pt/~alex/G4Classes/Examples
    and download **BraggPeak.zip**
  - Save it to your working area
    - if you see directories from last year, delete them first to avoid confusion
  - Use the file explorer to unzip the file, or open the Terminal and type
    - *cd Downloads/* — replace Downloads with the name of your folder
    - *unzip BraggPeak.zip*
    - *cd BraggPeak*
  - BraggPeak.cc has the main code of the simulation (inside the ***main()*** function)
  - open this file and let’s have a quick look at it
  - there are also 2 folders: ***include*** and ***src***
  - Both seem to have the same files: ***DetectorConstruction***, ***PhysicsList***, ***PrimaryGeneratorAction***, ***SteppingAction***
  - Actually, in the files inside *include* we **declare** variables and methods;
    in the files inside *src* we **implement** the actual code

# Concept of ‘event’ and ‘run’

- An **event** is what happens each time you run the simulation
  1. start with the primary particles defined by the user
  2. these are tracked until they stop, possibly interacting with materials
  3. if secondary particles are created (*e.g.* electrons after photoelectric effect), they are also tracked until they stop
  4. during this process we can extract useful information
     (deposited energy, interaction positions, etc.)
  5. a single event **will not** give us a good approximation of the overall response of the system
  6. must have many events under the same conditions — this is a **run**


arXiv:1303.2160 [nucl-ex]

# Concept of ‘event’ and ‘run’

- A **run** is a collection of events
    - sharing the same geometry
    - with the same physics conditions
    - primary particles are generated in the same way
    - ideally it should have enough events to be a good approximation of the response of the system being studied

## Mandatory classes

# *DetectorConstruction()*

Where we “construct” our detector

- Define the **materials** we want to use
- Define **geometric solids** needed for the geometry
- **Position** these solids in our virtual “laboratory”
  and associate a material to each
- If needed, define surfaces between solids
  (for optical processes only: reflection, refraction)
- Define visualisation properties of each element
  (colour, transparency)

# Materials

- ➡ We may define a material from its basic constituents:
  - Isotopes → Elements → Molecules, compounds and mixtures
- ➡ Or create materials from ‘scratch’
  (main properties only: atomic number, density, molar mass)
- ➡ Use GEANT4 predefined materials (from NIST database)
- ➡ The way you create a material depends on the physics
  you want to simulate
  - Some processes depend on the isotopic composition (e.g. neutron capture), but most do not (e.g. electromagnetic processes)
- ➡ Materials have properties
  (their use also depends on the specific process):
  - density, state, pressure, temperature

# Defining materials…

- From a single element:

```
G4double density = 1.390*g/cm3;
G4double a = 39.95*g/mole;
G4double z = 18.;
G4Material* lAr =
        new G4Material("LiquidArgon", z, a, density);
```

## Defining materials…

- **Molecules** are defined from individual elements (in this case we use the **number of atoms**)

```
a = 1.01*g/mole;
G4Element* elH  =
   new G4Element("Hydrogen",symbol="H",z=1.,a);
a = 16.00*g/mole;
G4Element* elO  =
   new G4Element("Oxygen",symbol="O",z=8.,a);

density = 1.000*g/cm3;
G4int components = 2;
G4Material* H2O =
   new G4Material("Water", density, components);
G4int natoms;
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);
```

# Defining materials…

- For compounds we use **mass fraction**

```
a = 14.01*g/mole;
G4Element* elN  =
   new G4Element(name="Nitrogen",symbol="N",z= 7.,a);
a = 16.00*g/mole;
G4Element* elO  =
   new G4Element(name="Oxygen",symbol="O",z= 8.,a);

density = 1.290*mg/cm3;
G4int components = 2;
G4Material* Air =
   new G4Material(name="Air", density, components=2);
G4double fracMass;
Air->AddElement(elN, fracMass=70.0*perCent);
Air->AddElement(elO, fracMass=30.0*perCent);
```

$\}$ **The sum must be 100%**

# Defining materials…

- We may also define mixtures, using existing materials or elements (in this case we also use **mass fraction**)

```
G4Element* elC  = …;   // define “carbon” element
G4Material* SiO2 = …;  // define “quartz” material
G4Material* H2O = …;   // define “water” material

density = 0.200*g/cm3;
G4Material* Aerog =
   new G4Material(“Aerogel", density, ncomponents=3);
Aerog->AddMaterial(SiO2, fractionmass=62.5*perCent);
Aerog->AddMaterial(H2O , fractionmass=37.4*perCent);
Aerog->AddElement (elC , fractionmass=0.1*perCent);
```

# Materials

- From GEANT's internal database (NIST materials)
- Full list of available materials can be found here:
  http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/Appendix/materialNames.html
- Use of "standard" materials makes comparison with results from other people easier
- Includes standard materials for medical use
  (e.g. G4_A-150_TISSUE, G4_ADIPOSE_TISSUE_ICRP, G4_B-100_BONE, etc.)

```
#include “G4NistManager.hh"

G4NistManager* man = G4NistManager::Instance();
  // Define material Air from the NIST database
  G4Material* Air  = man->FindOrBuildMaterial("G4_AIR");

  // Define material Tissue from the NIST database
  G4Material* Tissue  = man->FindOrBuildMaterial("G4_MUSCLE_SKELETAL_ICRP");
```

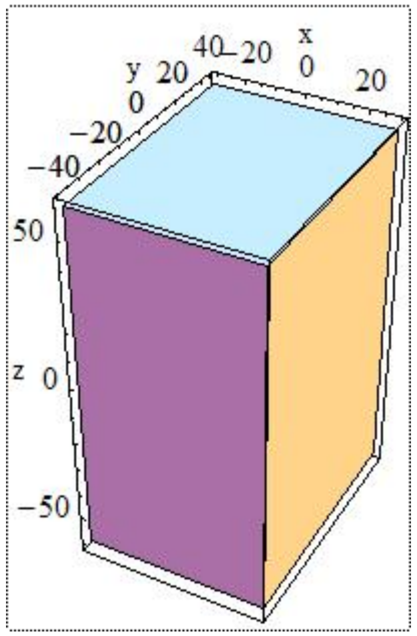
# Geometry

- To generate each element in the geometry of our experiment, we need **three** “layers”:
  1. **Solid** (defines the shape and the size)
  2. **Logical volume** (associates a material, adds visualisation properties)
  3. **Physical volume** (positions and rotates in the reference frame)

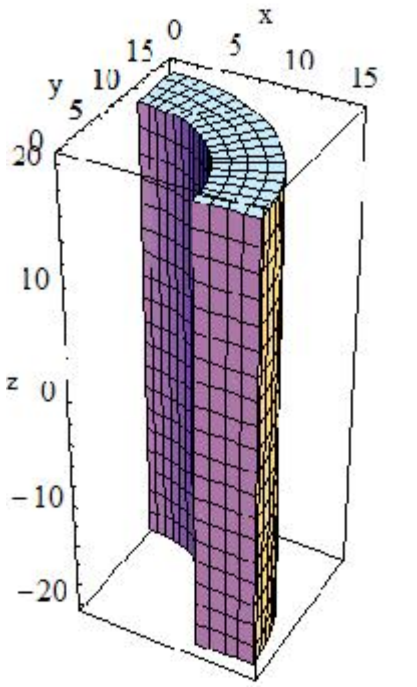
```
G4VSolid* pBoxSolid = new G4Box(“aBoxSolid”, 1.*m, 2.*m, 3.*m);

G4LogicalVolume* pBoxLog =
  new G4LogicalVolume( pBoxSolid, pBoxMaterial, “aBoxLog”, 0, 0, 0);

G4VPhysicalVolume* aBoxPhys =
  new G4PVPlacement( pRotation, G4ThreeVector(posX, posY, posZ),
                     pBoxLog, “aBoxPhys”, pMotherLog, 0, copyNo);
```

# Most common solids

```
G4Box(const G4String &pname,   // name
            G4double half_x,   // X half size
            G4double half_y,   // Y half size
            G4double half_z);  // Z half size
```

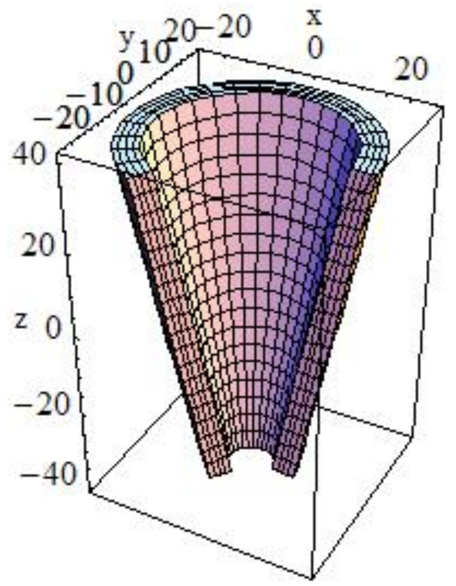


```
G4Tubs(const G4String &pname,  // name
             G4double  pRmin,  // inner radius
             G4double  pRmax,  // outer radius
             G4double  pDz,    // Z half length
             G4double  pSphi,  // starting Phi
             G4double  pDphi); // segment angle
```

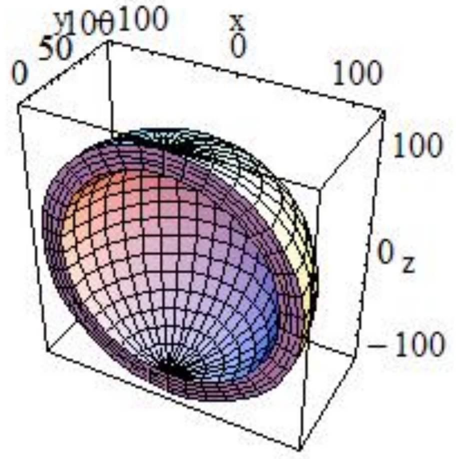


# More solids…

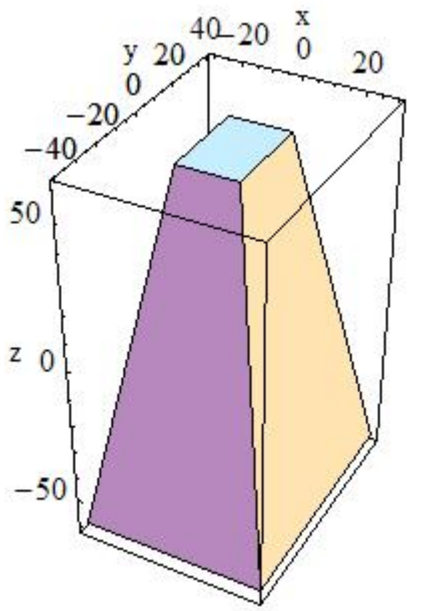
- Conical section - *G4Cons(…)*



- Sphere (or spherical shell) - *G4Sphere()*



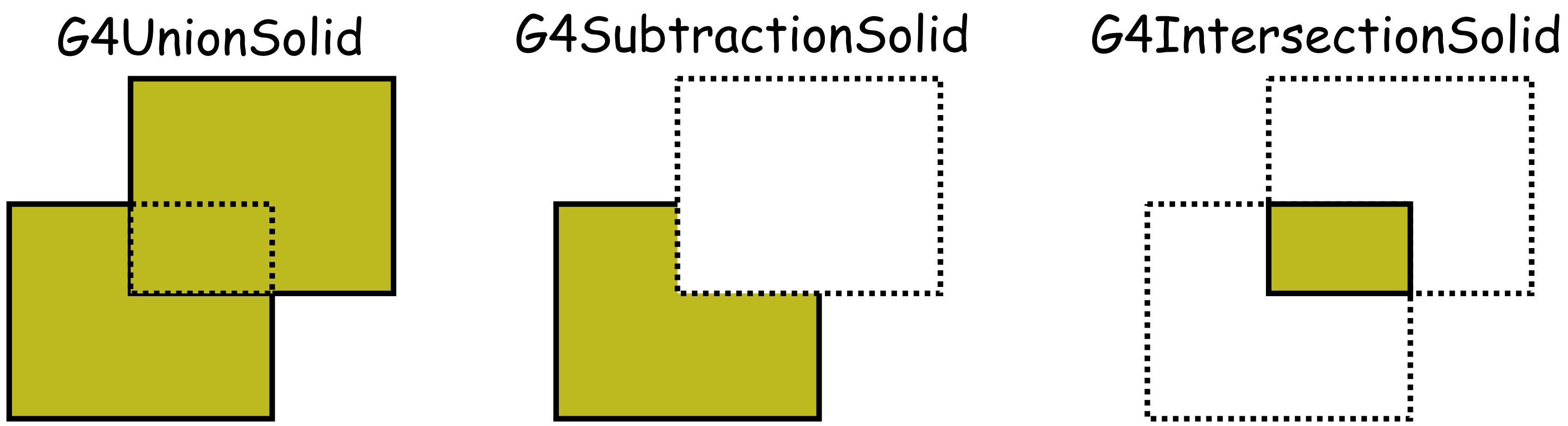
- *Pyramid* - *G4Trd()*



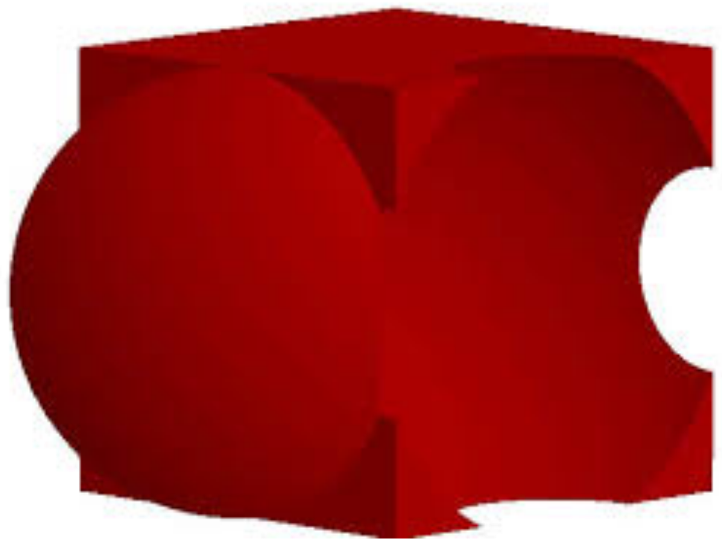
See all available, and their required parameters here

# Operations with solids

- It's possible to combine previously defined solids:
  - ➡ Union
  - ➡ Subtraction
  - ➡ Intersection



- This allows us to create more complex solids

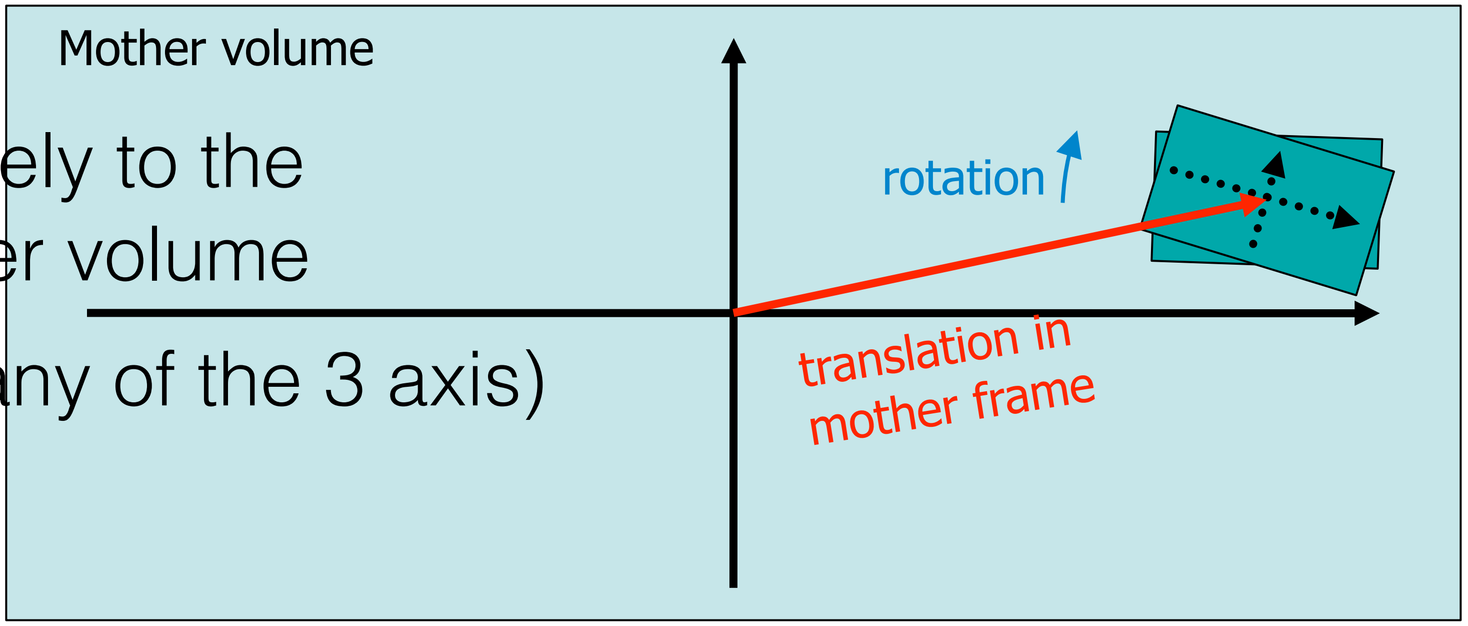


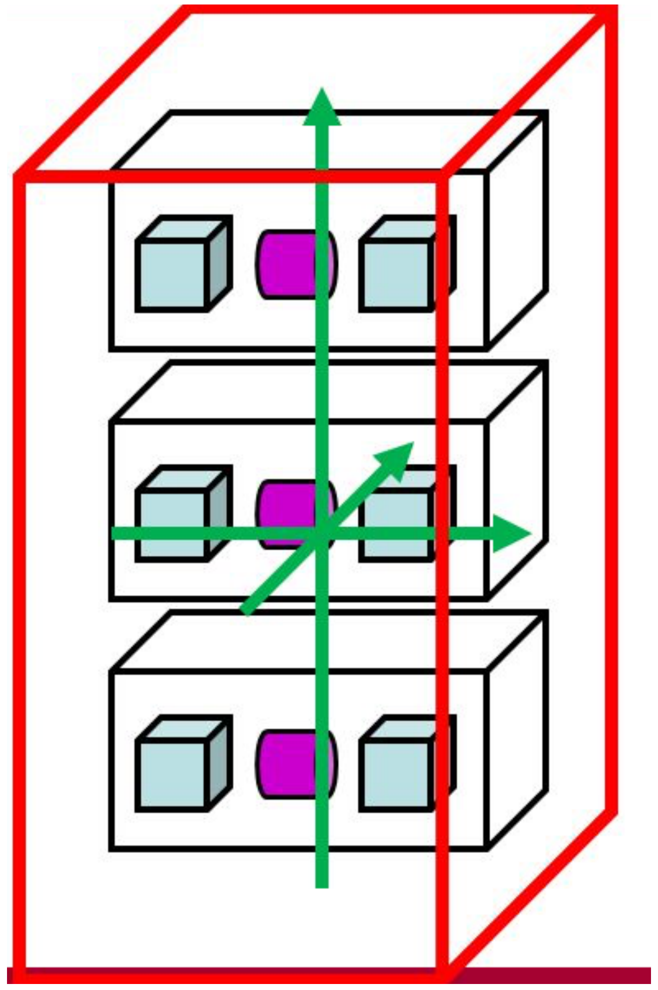
# Concept of hierarchy in the geometry

- A new volume has a **mother volume** and is placed relatively to its reference frame
- There is a special volume, called **‘world’**, which represents the ‘laboratory’
  - ➡ A volume **may not** extend beyond the limits of its mother volume
  - ➡ Volumes **must not** intersect each other
    (except within the mother-daughter paradigm)
- If any of these happens, the tracker (which follows each particle) may get confused and you may get unexpected/inconsistent results
- A logical volume may be placed several times in the geometry
  (including in different mother volumes)
  - each one becomes an independent geometry element

# Positioning

- Volumes are positioned relatively to the coordinate system of its mother volume
- They may also be rotated (in any of the 3 axis)





- If a mother volume is placed more than once, all daughters will appear in all of them
- The world volume is unique: it must be created first, and must fully contain all other volumes
- A global coordinate system is associated with the world volume

# Geometry example

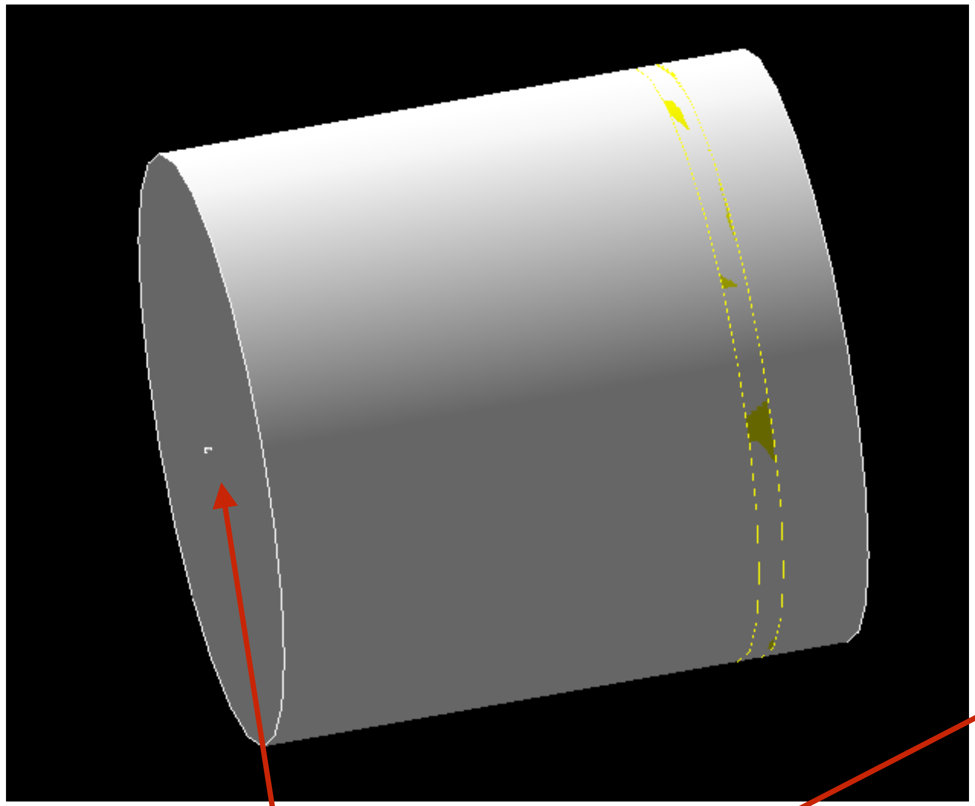
- To keep things more organised, create a work folder somewhere (e.g. *mkdir ~/Desktop/geant4*)
- Go back to the website and download B2a.zip
- Save it in your work folder
- On the terminal, navigate to inside the relevant folder *(e.g. cd ~/Desktop/geant4/B2a)*
- You may need to run this, but try to compile the simulation first (next slide)
  *source /usr/Geant4/geant4.10.04/share/Geant4-10.4.0/geant4make/ geant4make.sh*

# Geometry example

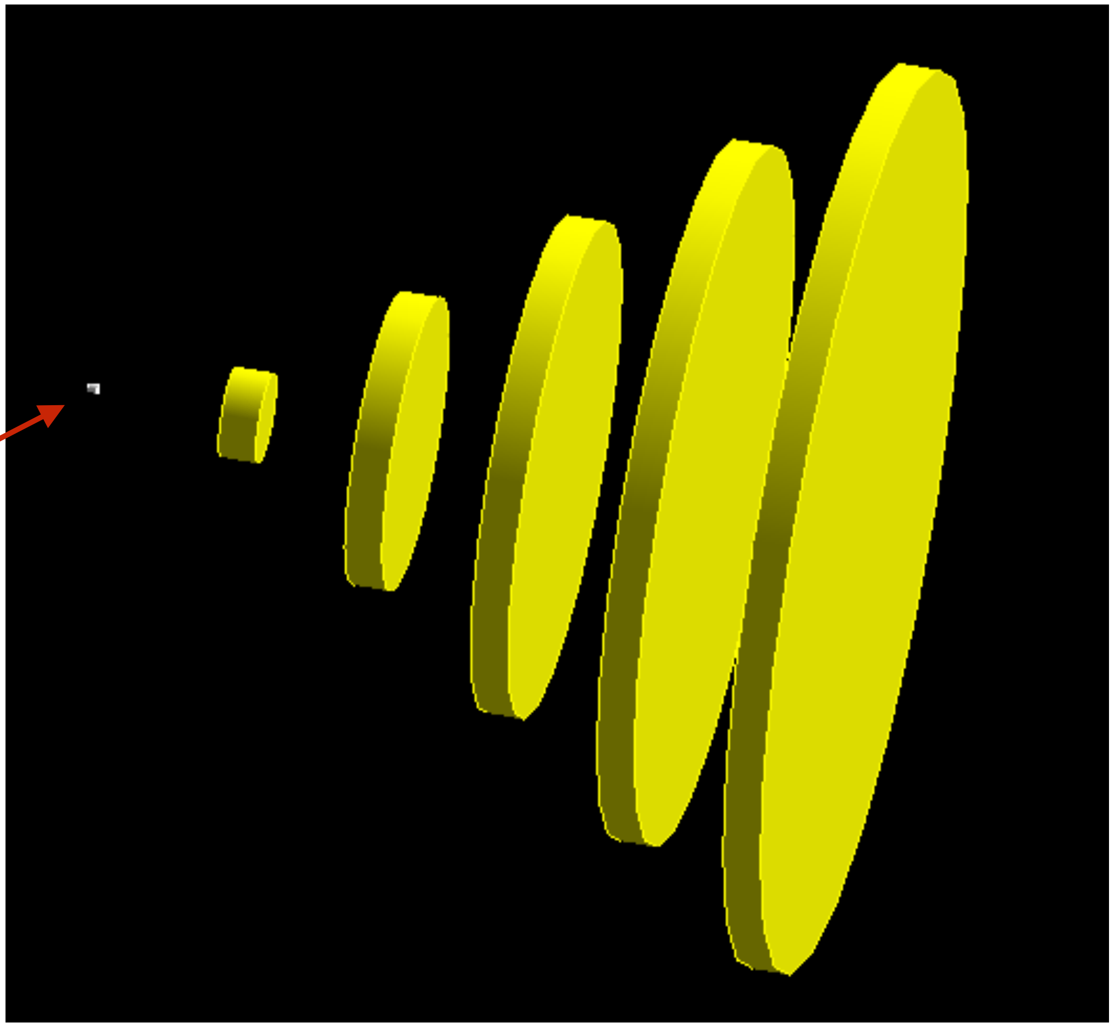
- Let’s compile our first simulation! — this is actually a GEANT4 example
  (type *make clean*, then *make*)
- Visualise the geometry by running the simulation
  (type *exampleB2a*)
- Rotate the geometry, zoom in/out using your mouse
- Type *exit* in the lower text box to quit GEANT4

# Geometry elements

Tracker (air)



Target (Pb)



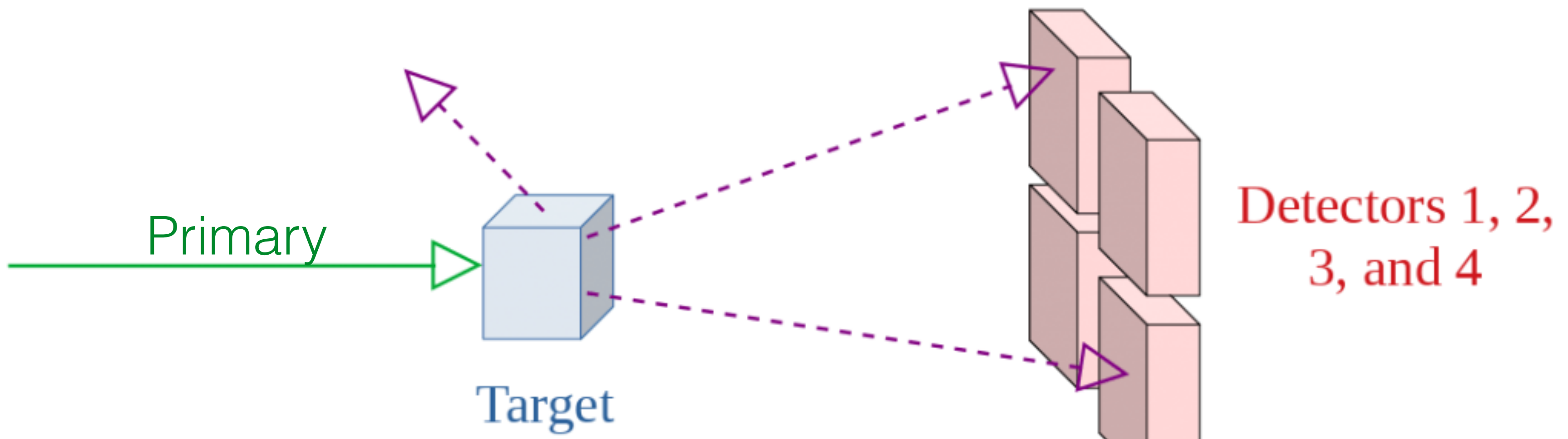
Tracker segments (Xe gas)

## (More advanced) Geometry examples

- Download the .wrl files from the website and explore the geometries
  - Underground physics
  - Human phantoms
  - Space telescopes

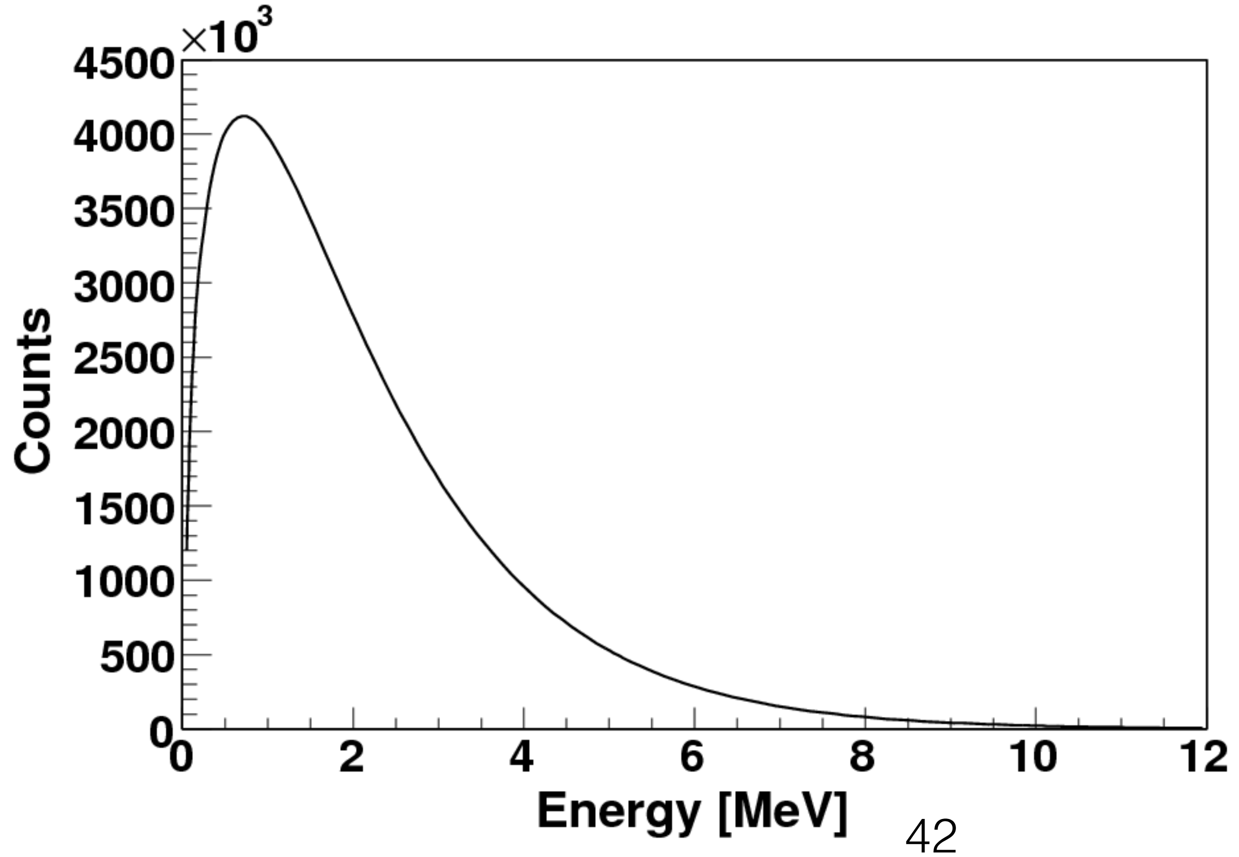
# Primary particle generator

- The initial particles in each event are called primaries (at least one, but we can have many)
- Must be defined by the user:
  - particle type (electron, proton, gamma, etc.)
  - initial position
  - starting energy
  - initial direction



# Primary particle generator

- Each of these properties may be constant, or generated using an algorithm
  - randomly
    - uniform position distribution inside a volume
    - isotropic direction
  - sampled from a distribution (*e.g.* with the rejection method)



Energy distribution of neutrons from a $^{252}$Cf source

# Primary generator example

- Open file B2aPrimaryGeneratorAction.cc
- Analyse how primary particles are defined
- What is the primary particle?
- Visualise particle tracks inside the geometry
  - run *exampleB2a* again
  - when geometry shows up, type ***/run/beamOn 10***
    - this will generate 10 events, starting with the defined primary particle
  - we can see **particle trajectories** and **hits/interactions**
- Change initial direction and/or energy
  - type *exit* to quit the simulation
  - change the code, save, and type *make* on the Terminal
  - run *exampleB2a*, type */run/beamOn 10* on the prompt
- Generate a different primary particle
  (mu-, e-, e+, neutron, gamma)
- Bonus: try changing the magnetic field value to 0.2 Tesla along *x* (line 306 in DetectorConstruction), recompile and run
  - *G4ThreeVector(0.2*tesla, 0., 0.)*

# Exercise 4 - Homework

- Define an isotropic generator
  - returns the components of a unitary vector *(x,y,z)* in each iteration
  - every spacial direction must have the same probability of being selected

# Installing GEANT4 on your laptop

- Direct installation on the system (Linux, macOS or Windows)
  - Useful if you want to continue using GEANT4 in the future (e.g. for your thesis)
  - It will likely be trickier, dependencies must also be installed, etc.
  - I have no experience at all with the Windows installation
  - Instructions on the GEANT4 web page
- Using docker and a Linux image with C++/GEANT4/Root
  - Should be faster and easier, although in some systems it may not be straightforward, there may be issues with visualisation
  - Instructions in these slides