Geant 4

Interacting with the Geant4 Kernel

Anton Lechner, CERN

Acknowledgements:
J. Apostolakis, G. Cosmo, M. Asai, A. Howard

http://cern.ch/geant4

I. Review of User Action Classes

Setting up a Geant4 application

- Mandatory classes of a Geant4 application:
  - Geometrical setup:
    - Volumes, materials, ...
    - *Derive your own class from G4VUserDetectorConstruction*
  - Physics setup
    - Particles, processes, models, production threshold,...
    - *Define a subclass of G4VUserPhysicsList (Note: G4VModularPhysicsList derives from this class)*
  - Primary track generation:
    - Primary particles, vertices, ...
    - *Derive your own class from G4VUserPrimaryGeneratorAction*

Setting up a Geant4 application

- Mandatory classes of a Geant4 application (cont.):
  - main() function of your application:
    - Create instances of the mandatory classes
    - Then, register the initialization classes, i.e. the concrete subclasses of G4VUserDetectorConstruction and G4VUserPhysicsList, to the run manager using the SetUserInitialization() function
    - Register the user action class, i.e. the concrete subclass of G4VUserPrimaryGeneratorAction, to the run manager using the SetUserAction() function

Example:

```
// main():

#include “G4RunManager.hh“
#include “MyDetectorConstruction.hh“
#include “MyPhysicsList.hh“
#include ”MPrimaryGenerator.hh“

main() {

  G4RunManager* runManager = new G4RunManager();

  // mandatory initialization classes:
  G4VUserDetectorConstruction* detector =  new MyDetectroConstruction();
  runManager -> SetUserInitialization(detector);

  G4VUserPhysicsList* physicsList = new MyPhysicsList();
  runManager -> SetUserInitialization(physicsList);

  // mandatory user action class:
  G4VUserPrimaryGeneratorAction* primaryGenerator =
                                          new MyPrimaryGenerator();
  runManager -> SetUserAction(primaryGenerator);
  ...
}
```

Optional user action classes

- Optionally, you can define additional user action classes:
  - Define your actions by deriving concrete classes from:
    - G4UserRunAction
    - G4UserEventAction
    - G4UserStackingAction
    - G4UserTrackingAction
    - G4UserSteppingAction
  - As for the mandatory classes, you also need to instantiate them in the main() function of your application and set them to the run manager by using the SetUserAction() function

Geant4 Terminology

- Brief overview
  - This slide will be shown again in the second part of the presentation

	Object	Description
Run	G4Run	Largest unit of simulation, that consist of a sequence of events: If a defined number of events was processed a run is finished.
Event	G4Event	Basic simulation unit in Geant4: If a defined number of primary tracks and all resulting secondary tracks were processed an event is over.
Track	G4Track	A track is NOT a collection of steps: It is a snapshot of the status of a particle after a step was completed (but it does NOT record previous steps). A track is deleted, if the particle leaves world, has zero kinetic energy, ...
Step	G4Step	Represents a particle step in the simulation and includes two points (pre-step point and post-step point).

Examples of usage (1/2)

- G4UserRunAction:
  - Has two methods (BeginOfRunAction and EndOfRunAction), which are invoked at the beginning and end of a run:
    - Can be used e.g. to initialize, analyse or store histograms, ...
  - Users can instantiate customized, user-defined run classes in the method GenerateRun
- G4UserEventAction:
  - Has two methods (BeginOfEventAction and EndOfEventAction), which are invoked at the beginning and end of an event:
    - E.g. one can apply an event selection in the beginning of a step, process information of hits in the end of an event, ...

Examples of usage (2/2)

- G4UserStackingAction:
  - Classify priorities of tracks
- G4UserTrackingAction:
  - Has two methods PreUserTrackingAction and PostUserTracking, which can be overloaded:
    - Can be used e.g. define trajectories, decide if trajectory should be stored, ...
- G4UserSteppingAction:
  - Has a method, UserSteppingAction, which is invoked in the end of an event:
    - E.g. you may change the track status in this method, ...

II. Retrieving information from various objects

Retrieving information from tracks and steps

Accessing information of steps and tracks
as the particle is propagated through a medium

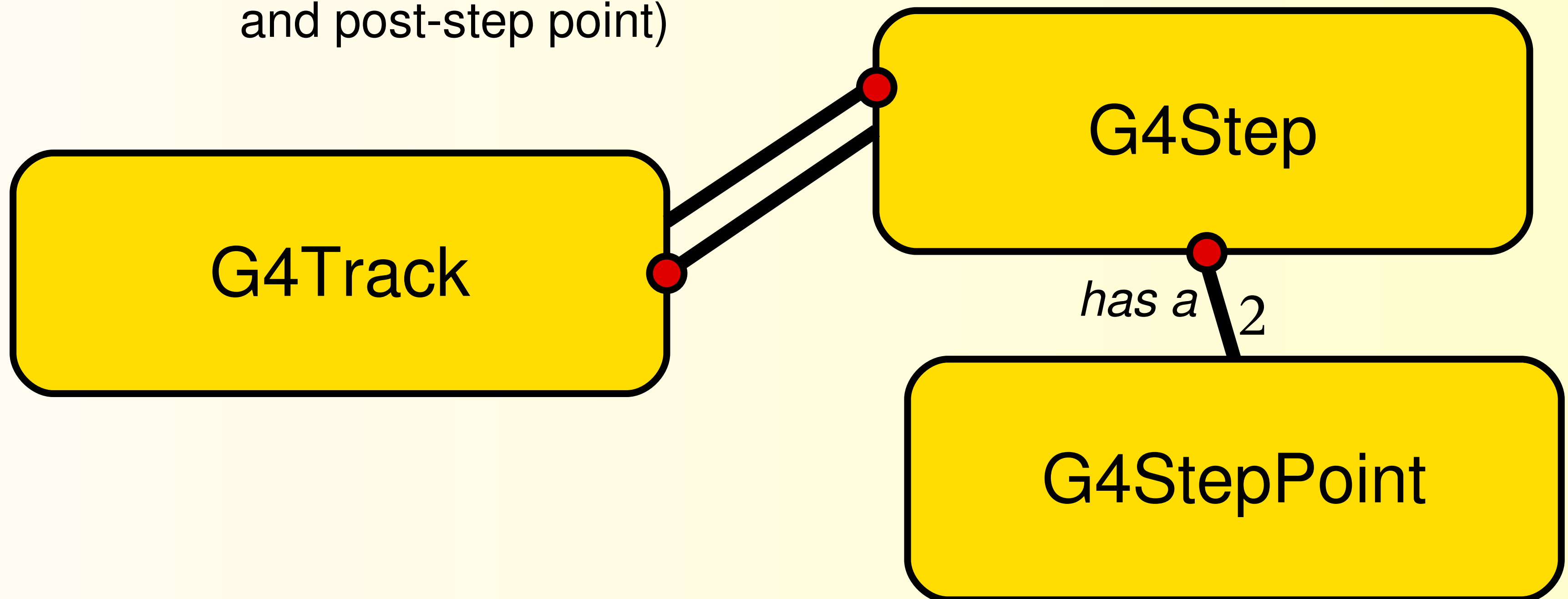
Geant4 Terminology

- Overview of important terms and definitions:

	Class	Description
Run	G4Run	Largest unit of simulation, that consist of a sequence of events: If a defined number of events was processed a run is finished.
Event	G4Event	Basic simulation unit in Geant4: If a defined number of primary tracks and all resulting secondary tracks were processed an event is over.
Track	G4Track	A track is NOT a collection of steps: It is a snapshot of the status of a particle after a step was completed (but it does NOT record previous steps). A track is deleted, if the particle leaves world, has zero kinetic energy (and no at-rest process occurs), ...
Step	G4Step	Represents a particle step in the simulation and includes two points (pre-step point and post-step point).
Step point	G4StepPoint	Represents an endpoint of a particle step

Class association

- Association between G4Track, G4Step and G4StepPoint:
  - Bi-directional association between G4Step and G4Track
  - G4Step has pointers to two G4StepPoint objects (pre-step and post-step point)



Endpoint of particle step

- The G4StepPoint class represents the endpoint of a particle step and contains (among other things):
  - Geometrical/Material information:
    - The coordinates of the particle position
    - The pointer to the physical volume that contains the position
    - The pointer to the material associated with this volume
  - Step status
  - Physics process information:
    - Pointer to physics process that defined step-length of current and previous step
  - And more information, which is used to update the G4Track object by G4Step, which contains the endpoint

Particle step (1/5)

- G4Step represents a step a particle is propagated in the simulation
- A G4Step object stores *transient* information of the step
  - In the tracking algorithm, G4Step is updated each time a physics process was invoked (for all along-step, post-step and at-rest actions of a step)
  - You can extract information from a step after the step was completed:
    - Both, the ProcessHits() function of your sensitive detector and UserSteppingAction() of your step action class receive a pointer to the G4Step object
    - Typically, you may retrieve step information in these functions e.g. to fill hit objects in ProcessHits(), ...

Particle step (2/5)

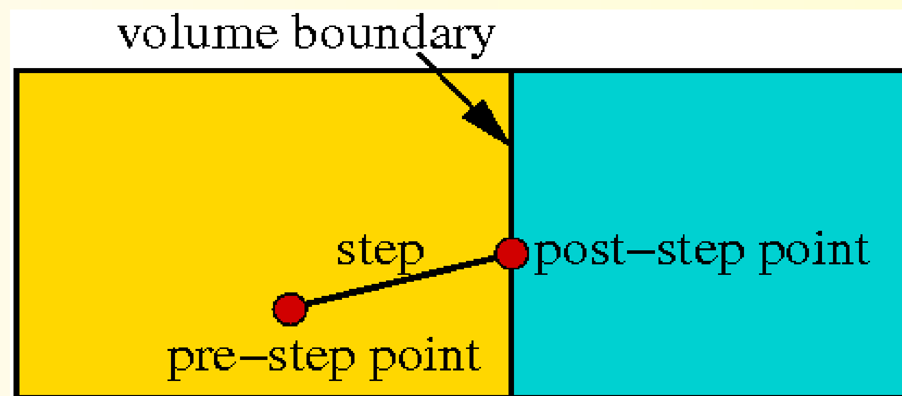
- A G4Step object contains:
  - The two endpoints, i.e. the pre-step and post-step point
    - Hence, one has access e.g. to the volumes, containing the step endpoints
  - Changes in particle properties between the points
    - E.g. difference of particle energy and momentum, ...
  - More step-related information like
    - Energy deposition on step, step length, time-of-flight, ...
  - A pointer to the associated G4Track object
- G4Step provides various Get methods to access these information or object instances:
  - E.g. G4StepPoint* GetPreStepPoint(), G4double GetStepLength(), ...

Example: Step information in SD

```
// in source file of your sensitive detector:


MySensitiveDetector::ProcessHits(G4Step* step,
                                 G4TouchableHistory*) {

   // Total energy deposition on the step (= energy deposited by energy loss
   //  process and energy of secondaries that were not created since their
   // energy was < Cut):
   G4double energyDeposit = step -> GetTotalEnergyDeposit();

   // Difference of energy , position and momentum of particle between pre-
   // and post-step point
   G4double deltaEnergy = step -> GetDeltaEnergy();
   G4ThreeVector deltaPosition = step -> GetDeltaPosition();
   G4double deltaMomentum = step -> GetDeltaMomentum();

   // Step length
   G4double stepLength = step -> GetStepLength();

}
```

Particle step (3/5)

- Step-points and geometrical boundaries:
  - If a step is limited by a boundary the post-step point is physically *on* the boundary
  - *Note, the post-step point is then considered to be in the next vol.*
  - This implies, the post-step point contains volume and material information of the next volume
    - Together with the content of the pre-step point object this allows to keep track of boundary processes



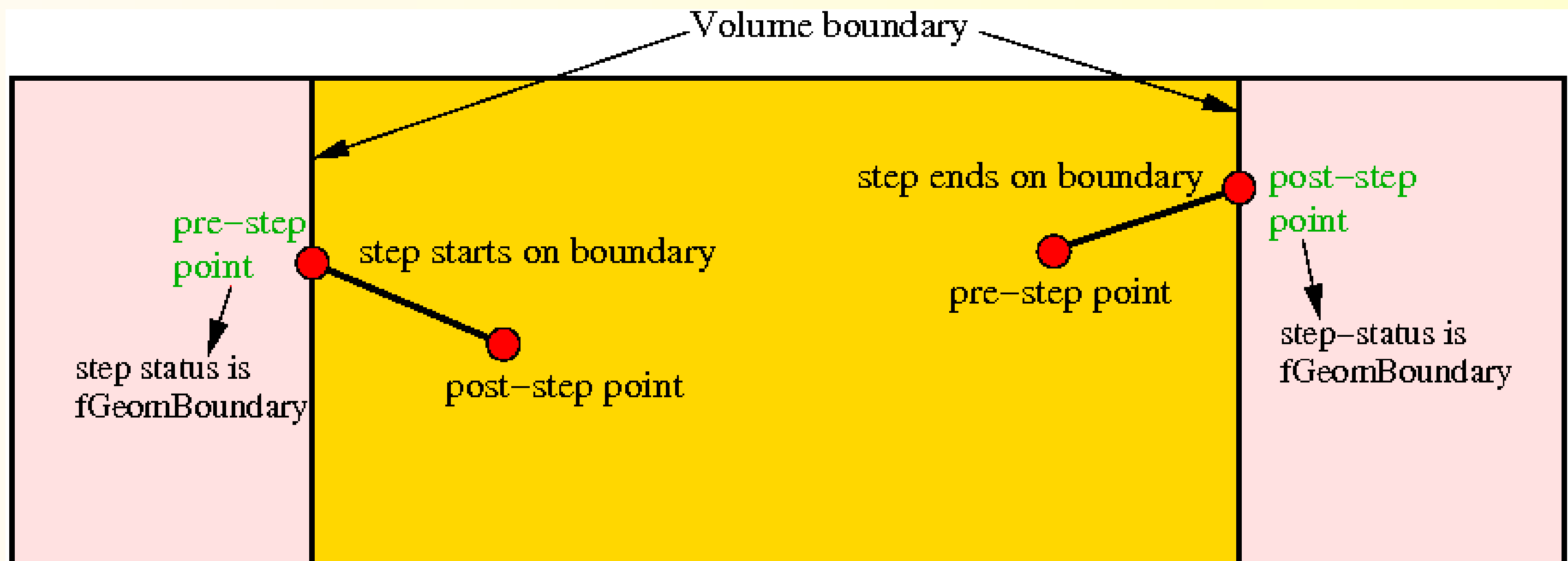
Particle step (4/5)

- Step-points and geometrical boundaries (cont.):
  - To check, if a step *ends* on a boundary, one may compare if the physical volumes of pre- and post-step points are equal, or one makes use of the step status:
    - The step status provides information about the process, that restricted the step length (see Appl. Developers Manual for details)
    - It is attached to step points: The pre-step point has the status of the *previous* step, and the post-step point of the *current* step
    - *If the status of the POST-step point is "fGeomBoundary", the step ends on a volume boundary (does not apply to world volume)*
  - To check if a step *starts* on a volume boundary, you can also use the step status:
    - *If the status of the PRE-step point is "fGeomBoundary", the step starts on a volume boundary (does not apply to world volume)*

See illustration on the next slide

Particle step (5/5)

- Step-points and geometrical boundaries (cont.):
  - Illustration of steps starting or ending on boundaries (steps are considered with respect to the yellow volume):



Example:

```
// in the source file of your user step action class:

#include "G4Step.hh"

UserStepAction::UserSteppingAction(const G4Step* step) {

  G4StepPoint* preStepPoint = step -> GetPreStepPoint();
  G4StepPoint* postStepPoint = step -> GetPostStepPoint();

  // Use the GetStepStatus() method of G4StepPoint to get the status of the
  // current step (contained in post-step point) or the previous step
  // (contained in pre-step point):
  if(preStepPoint -> GetStepStatus() == fGeomBoundary) {

      G4cout << "Step starts on geometry boundary" << G4endl;
  }
  if(postStepPoint -> GetStepStatus() == fGeomBoundary) {

      G4cout << "Step ends on geometry boundary" << G4endl;

      // You can retrieve the material of the next volume through the
      // post-step point :
      G4Material* nextMaterial = step -> GetPostStepPoint()->GetMaterial();
  }
}
```

Particle track (1/4)

- A Geant4 track, represented by G4Track, is NOT a collection of G4Step objects, but:
  - It is a “snapshot“ of the status of a particle *after a step was completed*, i.e. it has information corresponding the post-step point of the step
    - E.g. kinetic energy of particle, momentum direction, time since event and track was created, *track status (see later)*,...
    - It also contains a pointer to a dynamic particle class (see the next part of the presentation for details)
  - It does not record information of previous steps
  - However, it has also some information, that is not subject to changes during stepping: track ID, inform. about primary vertex,..
    - Primaries have track ID=1, secondaries have higher track ID's

Particle track (2/4)

- We learned that G4Track and G4Step have no memory of previous steps, and no G4Track object is available at the end of an event
  - However, if you activate the use of G4Trajectory objects, some track information
    - Then G4Trajectory objects will be available in G4Event at the end of an event
    - G4Trajectory has a collection of G4TrajectorPoint objects
    - G4Trajectory stores particular information of G4Tracks
    - G4TrajectoryPoints store particular information of G4Steps
  - Do not store too many trajectories (consume lots of memory, critical for some events)

Particle track (3/4)

- A track object is deleted if
  - the particle leaves the world volume,
  - it disappears due to a physical process (e.g. decay),
    - Note that in some hadronic interactions, the particle "looses" its identity: It is treated as a secondary particle due to fact that the interaction partners cannot be distinguished; in this case the primary track is deleted.
  - its kinetic energy falls to 0 (and no "AtRest" action is required),
  - the user kills the track.

Particle track (4/4)

- After each step the track can change its state
  - The status can be (states in red can only be set by the user, e.g. in user actions, but will not be set by the kernel):

Track Status	Description
fAlive	The particle is continued to be tracked
fStopButAlive	Kin. Energy = 0, but AtRest process will occur
fStopAndKill	Track has lost identity (has reached world boundary, decayed, ...), Secondaries will be tracked
fKillTrackAndSecondaries	Track and its secondary tracks are killed
fSuspend	Track and its secondary tracks are suspended (pushed to stack)
fPostponeToNextEvent	Track but NOT secondary tracks are postponed to the next event (secondaries are tracked in current event)

Examples:

```
// retrieving information from tracks (given the G4Track object "track"):

if(track -> GetTrackID() != 1)  {
    G4cout << "Particle is a secondary" << G4endl;

    // Note in this context, that primary hadrons might loose their identity
    if(track -> GetParentID() == 1)
                    G4cout << "But parent was a primary" << G4endl;

    G4VProcess* creatorProcess = track -> GetCreatorProcess();

    if(creatorProcess -> GetProcessName()  == "LowEnergyIoni") {
            G4cout << "Particle was created by the Low-Energy " <<
                    << "Ionization process" << G4endl;
    }
}
```

Example:

```
// in the source file of your user tracking action class:

#include "G4TrackingManager.hh"
#include "G4Electron.hh"

UserTrackingAction::PostUserTrackingAction(const G4Track* track) {


   // The user tracking action class holds the pointer to the tracking manager:
   // fpTrackingManager
   // From the tracking manager we can retrieve the secondary track vector,
   // which is a container class for tracks:
   G4TrackVector* secTracks = fpTrackingManager -> GimmeSecondaries();
   // You can use the secTracks vector to retrieve the number of
   // secondaries, the initial kinetic energies, the particle type, ...:
   if(secTracks) { ... }

   // Note: The G4TrackVector is defined as:
   // typedef std::vector<G4Track*> G4TrackVector;
   // and hence has all the functionalities of the vector container class of the
   // STL
}
```

Retrieving particle information

Accessing static and dynamic information of particles

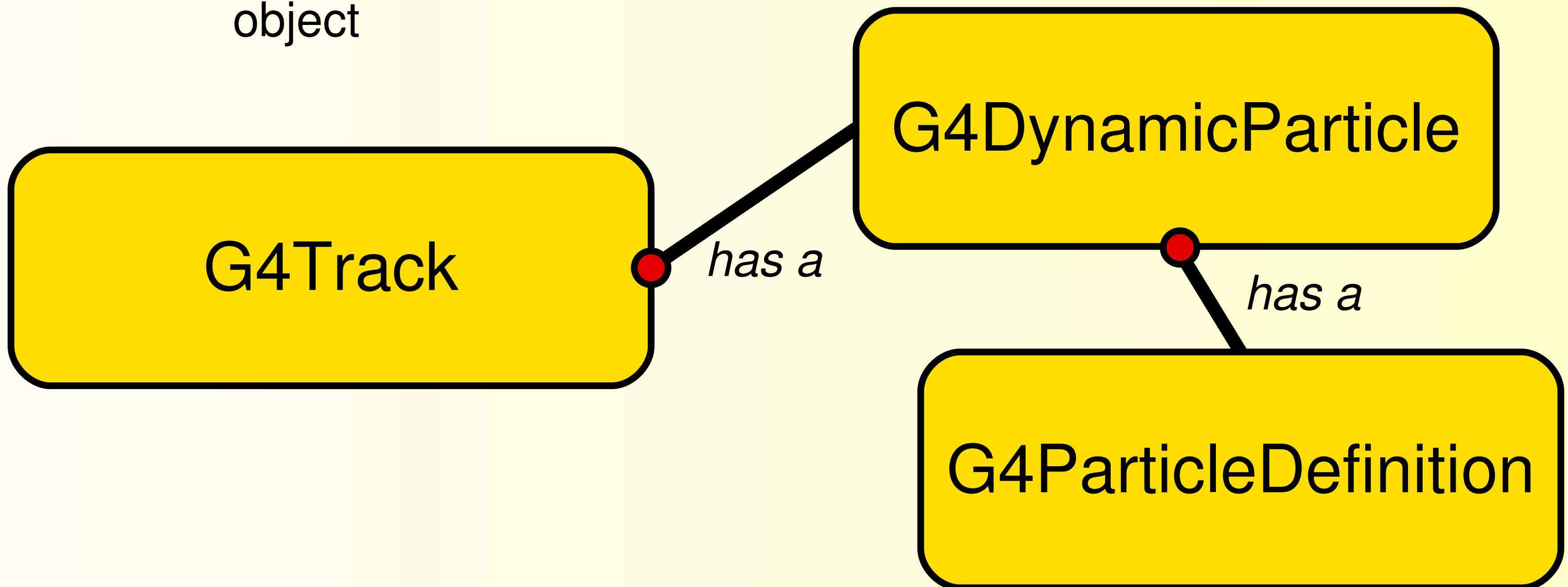
Geant4 Particle Terminology

- Overview of conceptual layers describing a particle:

Class	What does it represent?	What does it contain?
G4Track	Represents a particle that travels in space and time	Information relevant to tracking the particle, e.g. position, time, step,..., and *dynamic information*
G4DynamicParticle	Represents a particle that is subject to interactions with matter	Dynamic information, e.g. particle momentum, kinetic energy, ..., and *static information*
G4ParticleDefinition	Defines a physical particle	Static information, e.g. particle mass, charge, ... Also physics processes relevant to the particle

Class hierarchy

- Information of the other layers is contained through a “has-a” relation (aggregation):
  - G4Track has a pointer to a G4DynamicParticle object
  - G4DynamicParticle has a pointer to a G4ParticleDefinition object



Definition of particles (1/2)

- More than 100 types of particles are defined in Geant4
  - Particles are categorized into: leptons, mesons, baryons, bosons, short-lived and ions
- Most particles are represented by their own class that derives from G4ParticleDefinition (except ions)
  - E.g. G4Electron, G4Neutron, G4KaonPlus, ...
  - For each particle class, only a single static instance can exist
    - All G4DynamicParticle objects have a pointer to the same particle definition class if they deal with the same type of particle
    - The unique class instances are created (by the user) in the "initialization phase" (as a part of the Physics List set-up)
    - Once created, the user can retrieve information from the particle class instances using a range of Get-methods (see example page)

Definition of particles (2/2)

- The G4ParticleDefintion class contains:
  - Static information: Name, mass, charge, spin, life-time, ...
  - Pointer to G4ProcessManager, which contains a list of physics processes relevant to the particle
- Pointer to instances of particle definition classes:
  - Through G4DynamicParticle objects (see later)
  - The pointer to an instance of a particular particle class can be obtained through the static Definition() method
    - E.g. G4ParticleDefinition* electron = G4Electron::Definition();
  - Alternatively, the G4ParticleTable class (singleton) provides utility methods to find particles according to a specific attribute
    - E.g. by particle name: G4ParticleDefinition* electron = G4ParticleTable::GetParticleTable() -> FindParticle(“e-”);

Examples: Static particle information

```
#include “G4ParticleDefinition.hh“
#include “G4ParticleTable.hh“

G4ParticleDefinition* proton = G4Proton::Definition();

G4double protonPDGMass = proton -> GetPDGMass();
G4double protonPDGCharge = proton -> GetPDGCharge();

G4int protonPDGNumber = proton -> GetPDGEncoding();

G4String protonPartType = proton -> GetParticleType(); // “baryon“
G4String protonPartSubType = proton -> GetParticleSubType(); // “nucleon“

G4int protonBaryonNmb = proton -> GetBaryonNumber();


G4ParticleTable* ptable = G4ParticleTable::GetParticleTable();
G4ParticleDefinition* pionPlus = ptable -> FindParticle(“pi+”);

G4bool particleIsStable = pionPlus -> GetPDGStable();

G4double pionPlusLifeTime = pionPlus -> GetPDGLifeTime();

G4double pionPlusIsospin = pionPlus -> GetPDGIsospin();
```

Dynamic particles (1/2)

- A G4DynamicalParticle object represents an *individual* particle (whereas G4ParticleDefinition represents a particle type)
  - Each G4Track object has an unique instance of G4DynamicParticle, that exists as long as the track is not deleted
- A G4DynamicParticle object contains:
  - Dynamic information, i.e. physical properties that may change from step to step: kinetic energy, spin, polarization, charge (ions), ...
  - It holds a pointer to a particle definition object (static particle information)

Dynamic particles (2/2)

- Access to information of a dynamic particle:
  - Various Get methods are defined in G4DynamicParticle to allow the retrieval of dynamic information
    - Once all PostStepDoIt() methods were invoked (for a given step), the G4DynamicParticle instance is updated by the track to contain the particle properties resulting from the physics processes of the step.
    - Typically, you may retrieve dynamic information in the ProcessHits() function of your sensitive detector, or in UserSteppingAction() of your step action class.
  - The GetDynamicParticle() method of G4Track returns a pointer to the associated instance of G4DynamicParticle
  - Use the GetDefinition() method of G4DynamicParticle to obtain the pointer to the G4ParticleDefinition object.
    - Proceed as previously shown to retrieve static information

Example: Particle inform. from steps

```
#include “G4ParticleDefinition.hh“
#include “G4DynamicParticle.hh“
#include “G4Step.hh“
#include “G4Track.hh“

// Retrieve from the current step the track  (after PostStepDoIt of step is
// completed):
G4Track* track = step -> GetTrack();

 // From the track you can obtain the pointer to the dynamic particle:
const G4DynamicParticle* dynParticle = track -> GetDynamicParticle();

 // From the dynamic particle, retrieve the particle definition:
G4ParticleDefinition* particle = dynParticle -> GetDefinition();

 // The dynamic particle class contains e.g. the kinetic energy after the step:
G4double kinEnergy = dynParticle -> GetKineticEnergy();

 // From the particle definition class you can retrieve static information like
 // the particle name:
G4String particleName = particle -> GetParticleName();

 G4cout <<  particleName <<  “: kinetic energy of “
             <<  kinEnergy/MeV <<  “ MeV“
             <<  G4endl;
```