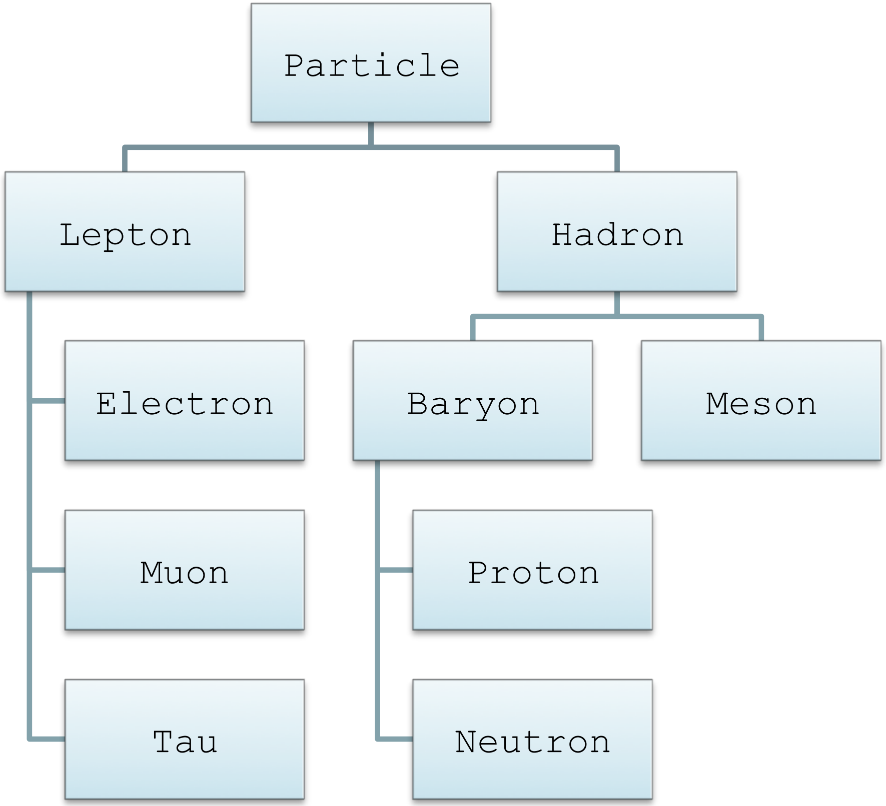
Scientific Programming Using Object-Oriented Languages
Module 6: Inheritance, Abstract Methods & Interfaces

Aims of Module 6:

- Recognize when it is appropriate to use an inheritance relationship between classes.
- Understand abstract classes and interfaces and when it is appropriate to use them.
- Understand the syntax of inheritance, abstract classes and methods, and interfaces.
- Have a deeper understanding of polymorphism attained through the use of superclasses and interfaces.

Inheritance

- A *subclass* **extends** a *superclass.*
- The subclass *inherits* all the public and protected member variables and methods of its superclass.
- Inheritance is a way of reusing the code in a class to represent a family of related classes.
- The hierarchy can be as deep as you like, but each class has only a single superclass in Java. (C++ is different).



Inheritance: Superclass

- Definition of superclass `Animal`

```
public class Animal {
  private String name;
  public void setName(String name) {this.name = name;}
  public String getName() {return this.name;}
  public void speak() {
    System.out.println("Makes a noise");
  }
}
```

Inheritance: Subclass

- Definition of subclass `Cat`

```
public class Cat extends Animal {
  @Override
  public void speak() {
    System.out.println("Meow");
  }
  public void purr() {
    System.out.println("RRrrRRrrRR");
  }
}
```

- `extends` is the keyword for inheriting from a superclass.
- A subclass can
  - override methods inherited from the superclass;
  - add methods to the superclass.

Inheritance: Subclass (Continued)

- The subclass `Cat` can do everything `Animal` can do and more.

```
Cat cat = new Cat();
cat.setName("Felix");
System.out.println(cat.getName());
cat.speak();
cat.purr();
```

- This is an example of the *Liskov Substitution Principle*.
  - Barbara Liskov (1987), formulated concisely by Barbara Liskov and Jeanette Wing (1994):
  - "Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T."
  - Less formally, from Martin Odersky: "If A <: B then everything one can do with a value of type B one should also be able to do with a value of type A."
- This enables a type of *polymorphism*…

Polymorphism

- Polymorphism (from Greek for *many forms)* means
  - "providing a single interface to entities of different types" (Bjarne Stroustrup)
  - a message can be passed to different objects, each responding appropriately.

- Using polymorphism we can write code that takes an `Animal` or `Collection` (or a `Collection` of `Animal`s) as input and leave the detailed implementation to the classes `Cat`, `Dog`, `Vector`, `HashSet`...

Inheritance: Polymorphism and Overriding

- The subclass `Cat` *overrides* the `speak` method in the superclass.
- We can use different types of `speak` method as if they were the same and each will use the appropriate method: this is *polymorphism*.

```
Animal[] animals = {new Cat(), new Dog()};
for (Animal a : animals) {
    a.speak();
}
```

- The version of the method used depends on the actual type (subclass) of the object: called “late binding”, “dynamic binding” or “run-time binding”.
- Static (class) methods and member variables *hide* superclass versions instead of overriding them: version accessed depends on the class it is accessed from.
- Must not to break the substitution principle e.g. overridden methods may not have more restrictive access.

Inheritance: Constructors

- Constructors are *not* inherited: you need to know the actual type of an object to create it.

```
Animal beast = new Animal();
Animal cat   = new Cat();
```

- The first line in a subclass constructor is a call to the superclass constructor:
  - `super()`, `super(int)`, etc. depending on version required
  - actually `super()` is automatically added by the compiler if you don't call a superclass constructor explicitly.

Inheritance Example: Superclass

- Take `SimpleCounter` from module 2 and extend it:

```
public class SimpleCounter {

    protected int counter;

    public SimpleCounter() {}
    public SimpleCounter(int val) { counter = val; }

    public void setCounter(int val) { counter = val; }
    public int getCounter() { return counter; }

    }
}
```

- We can create more specialized type of counter that reuse the existing code from `SimpleCounter` while adding further functionality.
- Note the use of the keyword **`protected`**…

Access Specifiers

- As well as `public`, `private` and default member variables and methods, we can now have `protected` ones.
- `protected` means accessible from subclasses as well as this class, even if the subclass is in a different package.

Accessible	Class	Package	Subclass	Anywhere else
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
(default)	Y	Y	N	N
`private`	Y	N	N	N

Inheritance Example: Subclass

- A counter with a limit:

```
public class LimitedCounter extends SimpleCounter {
   protected static int max = 1000;

   public LimitedCounter(int val) {
      super();
      if (val<=max) counter = val;
  }

   public void setCounter(int val) {
      if (val<=max) counter = val;
  }
}
```

- Substitution principle means that an overriding method cannot throw checked exceptions that are not declared in the superclass method.
- So here cannot throw checked exception if value out of range, unless we modify the superclass.
- Need some foresight when designing classes to be extended.

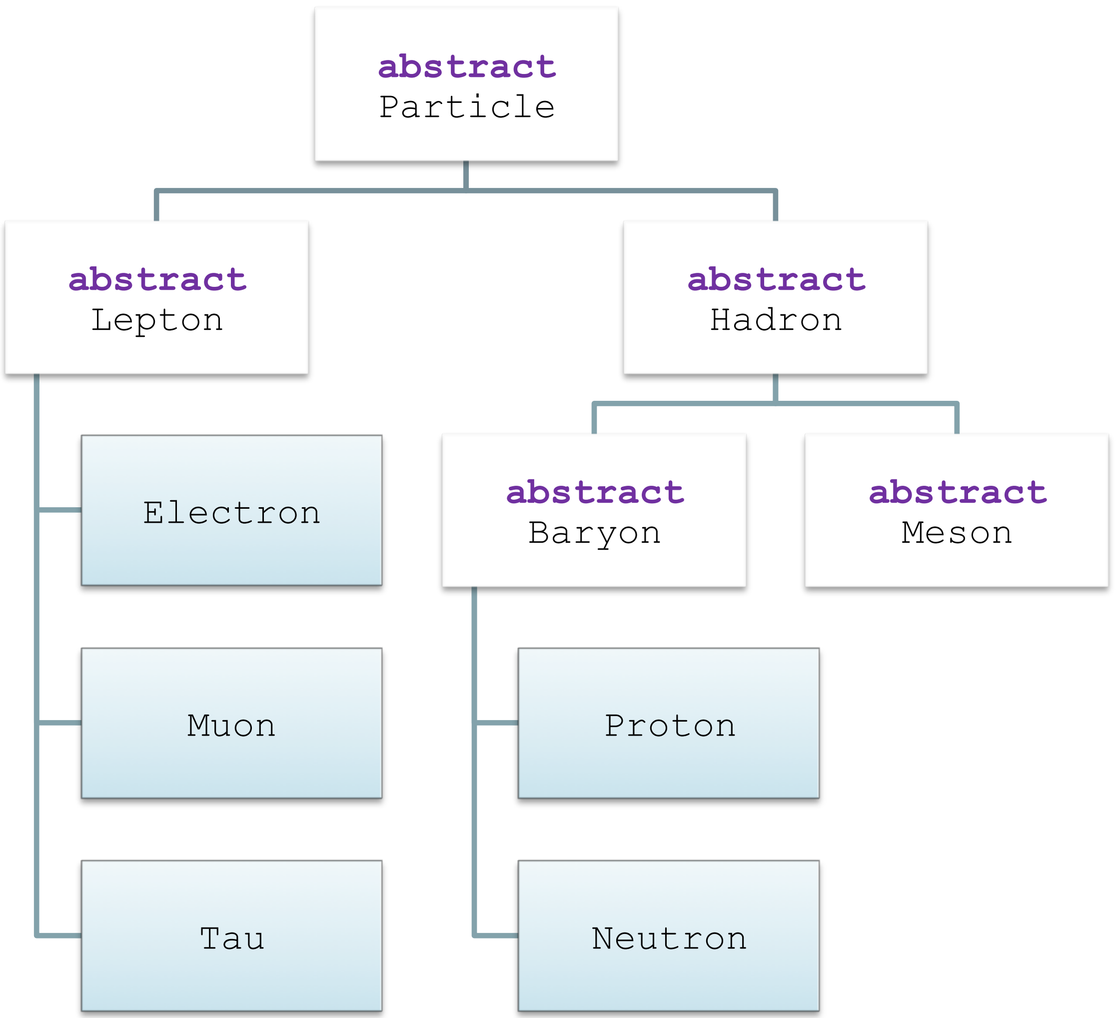
Abstract Classes

- Sometimes a superclass represents a general type of object that cannot be instantiated itself, and this can be represented by an *abstract* class.
- It doesn't really make sense to instantiate an Animal without knowing its actual type.

```
public abstract class Animal {
  private String name;
  public void setName(String name) {this.name = name;}
  public String getName() {return this.name;}
  public abstract void speak();
}
```

- Rather than define a default **`speak`** method that we never use, we can declare it **`abstract`** and leave it to each *concrete* subclass to define it appropriately.
- A *concrete* class is one that can be instantiated: it must define all methods that are left abstract by the superclass.

Abstract Classes: Physics Example



Abstract Classes: Example

- Abstract class definition:

Keyword for abstract class definition

```
public abstract class Particle {
    public abstract String name();
    // Other methods (abstract or concrete)
}
```

- An abstract class can extend another abstract class:

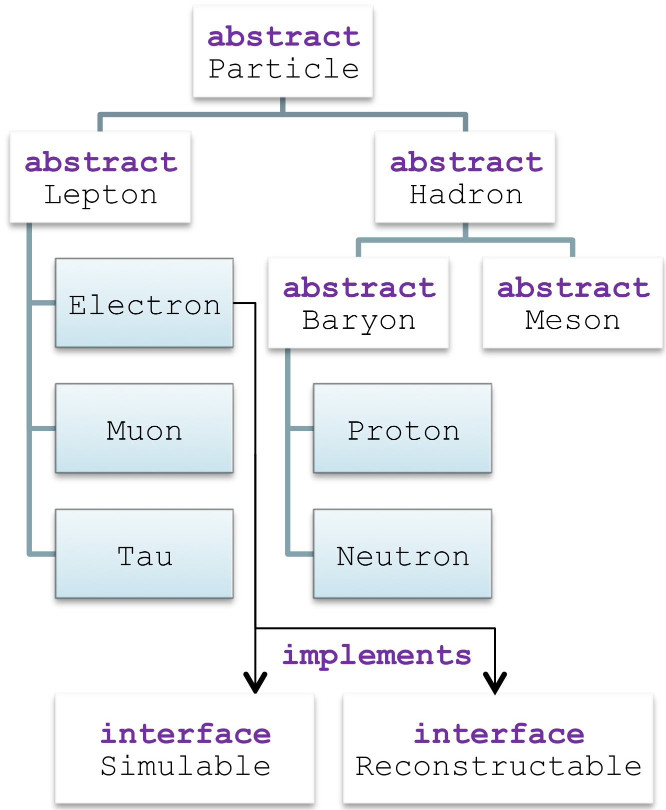
```
public abstract class Lepton extends Particle {
    public abstract int generation();
}
```

- A concrete class ***must*** implement all abstract methods:

```
public class Electron extends Lepton {
    public int generation() {
        return 1; // 1st lepton generation
    }
    public String name() {return "Electron";}
}
```

Interfaces

- Go beyond abstract classes: provide a set of method declarations but no implementation at all – an *interface*.
- May be *implemented* by any number of classes anywhere in the class hierarchy.
- A class can implement any number of different interfaces even though it can only extend one superclass.
- Implementing an interface is often likened to agreeing to a *contract*.
- The contract represented by an interface is given by
  - its methods, their arguments and return types;
  - a statement of how an implementation of each method is required to behave.
- The latter makes good, clear comments particularly important when creating an interface!



Interfaces: Example

- Interface to a `Simulable` collider object:

Keyword for interface definition

```
public interface Simulable {
    boolean depositsIonisationInTracker();
    boolean depositsEnergyInCalorimeter();
}
```

- Interface to a `Reconstructable` collider object:

```
public interface Reconstructable {
    double energyResolution();   // energy resolution in GeV
    double angularResolution(); // angular resolution in radians
}
```

Interfaces: Implementation Example

- The class `Electron`
  - **`extends`** the abstract class `Lepton`
  - **`implements`** both interfaces

Keyword for implementing interfaces

```
public class Electron extends Lepton
                      implements Simulable, Reconstructable {
   public int generation() {return 1;}
   public String name() {return "Electron";}
   public boolean depositsIonisationInTracker() {return true;}
   public boolean depositsEnergyInCalorimeter() {return true;}
   public double energyResolution() {return 0.01;}
   public double angularResolution() {return 0.1;}
}
```

Interfaces: Inheritance and Polymorphism

- An interface can extend any number of other interfaces
- An instance of a class can be accessed using any of the interfaces the class implements:

```
Electron e = new Electron();
Lepton l = e;
Simulable s = e;
Reconstructable r = e;
```

- This is polymorphism again, and is the main reason for using interfaces: your code can use objects without knowing their class, only that they implement a certain interface.

```
public ParticleTrack track(Reconstructable r) {
 // code using only methods of Reconstructable interface
}
```

- This enables great flexibility in applying the same code to different type of object. Interfaces are used extensively in the Java API, e.g. in the collections framework.

Collection Interfaces

- The Java Collections Framework makes good use of interfaces (and abstract classes).
- Main interfaces are
  - `Set`
  - `List`
  - `Queue`
  - `Map`

  `Set`, `List`, `Queue` **extends** `Collection`
- Examples for implementing classes:
  - `ArrayList` **implements** `List`
  - `HashSet` **implements** `Set`
  - `HashMap` **implements** `Map`
- Write code that refers to the most general Collections interface applicable to your problem, e.g.
  - Order of elements is arbitrary
    - `Collection` will work with `ArrayList`, `LinkedList`, `HashSet`
  - Order of elements is significant
    - `List` will work with `ArrayList`, `LinkedList`