Entities
students (records)
courses
↓
type hierarchy
no copies
no assignment
no default constructor
no type conversions
no equality / address equality
or ID equality

Values
grades
student ID
course ID
→ equality, comparison important
computation operators
final class, methods
immutable
lots of copying (deep)

CS 247: Software Engineering Principles

Value vs. Entity Objects, Information Hiding

Readings: Eckel, Vol. 1
Ch. 5 Hiding the Implementation

```
class student {
  public:
    student(const student&) =
        delete; //C++11
    student& operator=(const student&) =
        delete;
  private:
    student(const student &); //C++98
};
```

Entity vs. Value Objects

Entity Object

- computer embodiment of real-world entity
- each object has a distinct identity
- object with the same attribute values are NOT equal

Entity vs. Value Objects

Entity Object

- computer embodiment of real-world entity
- each object has a distinct identity
- object with the same attribute values are NOT equal

Entity ADT Examples

- physical objects: airplane, runway, taxiway, ...
- people: passenger, booking agent, ...
- records: customer information, boarding pass, flight schedule, ...
- transactions: reservations, cancellations, receipts, ...

Entity vs. Value Objects

Entity Object

- computer embodiment of real-world entity
- each object has a distinct identity
- object with the same attribute values are NOT equal

Entity ADT Examples

- physical objects: airplane, runway, taxiway, ...
- people: passenger, booking agent, ...
- records: customer information, boarding pass, flight schedule, ...
- transactions: reservations, cancellations, receipts, ...

Value Object

- simply represents a value of an ADT
- objects with the same attribute values are considered to be identical

Entity vs. Value Objects

Entity Object

- computer embodiment of real-world entity
- each object has a distinct identity
- object with the same attribute values are NOT equal

Entity ADT Examples

- physical objects: airplane, runway, taxiway, ...
- people: passenger, booking agent, ...
- records: customer information, boarding pass, flight schedule, ...
- transactions: reservations, cancellations, receipts, ...

Value Object

- simply represents a value of an ADT
- objects with the same attribute values are considered to be identical

Value ADT Examples

- mathematical types: rational numbers, polynomials, matrices, ...
- measurements: size, distance, weight, mass, energy, duration, ...
- other quantities: money
- other properties: colour, location, date, time, ...
- restricted value sets: names, addresses, postal codes, number ranges, ...

Design of Entity ADTs

An operation on an entity object should reflect a real-world event:

- copying an entity does is not meaningful; program no longer reflects reality; operations on copies are uncoordinated and can be lost (when copies disappear)
  - prohibit copy constructor
  - prohibit assignment
  - prohibit type conversions
  - avoid equality
  - clone operation may be useful

- computations on entities are not meaningful
  - think twice before overloading operators except `new` and `delete`
  - `operator<` is useful if overloaded to apply to entity's name or unique id

- entities referred by pointers (a consequence of the no-copy rule)

Design of Value-based ADTs

Equality is important in value types
- equality and other comparison operators
- copy constructor
- assignment operator

Computations involving values may make sense
- consider overloading arithmetic operators

Virtual functions and inheritance are uncommon

Mutable Objects

Mutable Value-based ADTs (e.g., Date) are problematic when they can be referenced from two variables

```
Person myPerson ( "David O'Leary", new Date(1, "May", 1990) );
// name, DOB
cout << myPerson.DOB() << endl;

Date myDate = myPerson.DOB();
myDate.monthIs( myDate.month() + 1 );
cout << myPerson.DOB() << endl;
```

Mutable vs. Immutable Objects

Entity Objects are ***mutable***

- their objects can change value via mutators, other functions

Value-based Objects are usually ***immutable***

- their objects cannot change value
- instead, variables of the ADT are assigned a different object

- no mutators
- member functions cannot be overridden (non virtual)
- copy/assignment operations are deep copies
- all data members are private
- all data members are of primitive or immutable types, else
    - make a copy of any (mutable) attribute parameter value
    - make a copy of any output (mutable) attribute return value

Singleton Design Pattern

Singleton design pattern - ensures that exactly one object of our ADT exists

```
class Egg {
  static Egg e;           // singleton instance
  int i;                  // data member
  Egg(int ii) : i(ii) {} // private constructor
public:
  static Egg* instance() { return &e; }
  int val() const { return i; }
  Egg(const Egg&) = delete;              // prevent copy
  Egg& operator= (const Egg&) = delete; // prevent assign
};

Egg Egg::e(42);           // initialization of singleton
```

Exposed Implementation

```
class Rational {
public:
    Rational (int numer = 0, int denom = 1);
    int numerator() const;
    int denominator() const;
private:
    int numerator_;
    int denominator_;
};
```

PImpl Idiom

Encapsulate the data representation in a nested private structure (which can be in a separate file).

```
class Rational {
public:
    Rational (int numer = 0, int denom = 1);
    int numerator() const;
    int denominator() const;
private:
    struct Impl;
    Impl* rat_;
public:
    ~Rational();
    Rational ( const Rational& );
    Rational& operator= ( const Rational& );
};
```