Reference counting (1)

- PETSc uses very simple **reference counting**
- when the `PetscObject` is **created** using `Create` function,
  its `PetscInt refct` field is set to 1
- one (typically higher) object can reference another (typically lower) object, e.g.
  when calling `KSPSetOperators(ksp,mat,...)`
  `KSP ksp` references `Mat mat`
- done by `PetscObjectReference` funtion
- each time an object **references** another object,
  the referencing object increments `refct`
  of the referenced object by 1

Reference counting (2)

- when the `PetscObject` is **destroyed** using `Destroy` function following rules apply
- if the `PetscObject` **is** `NULL` **nothing is done**
- If the `PetscObject` **is not** `NULL` and the **counter > 0**,
  then `Destroy`
  - **nullifies** the pointer
  - **decrements** the reference counter
- If the `PetscObject` **is not** `NULL` and **counter equals 0**,
  then `Destroy`
  - calls **type-specific** private **destroy routine**
  - **deallocates** the whole object

Reference counting (3)

- PETSc uses ***"destroy always"*** strategy
  - no overhead
  - you are obliged to call the destroy function each time you "leave" the `PetscObject`
    - it is not needed anymore in current scope
    - the referencing `PetscObject` is being deallocated
  - `(#PetscObjectReference calls) < (#Destroy calls)`
  - no problem with calling `Destroy` on `NULL` pointer (just no-op)
- Easier than pure C/C++ free/delete, harder than **smart pointers**
  (in new C++ standard, Boost, Trilinos RCP, etc.)
  that use ***"destroy never"*** strategy (automatic deallocation)