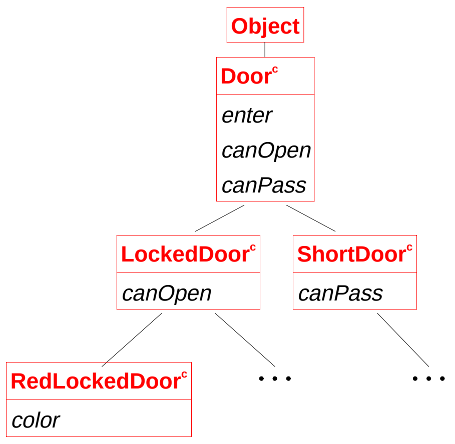
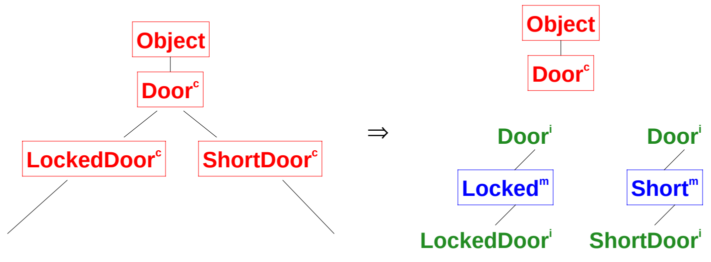
Classes



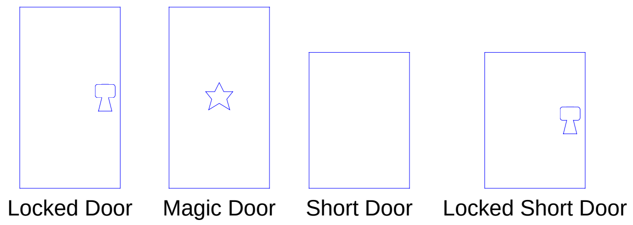
- Each node is a ***class extension***
- Each chain of nodes to the root is a ***class***

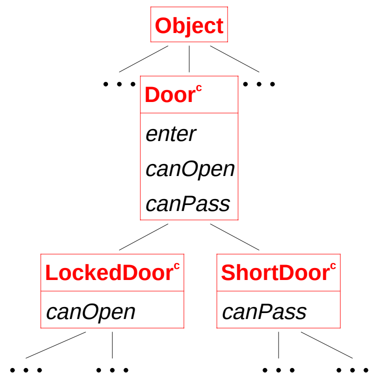
Mixins



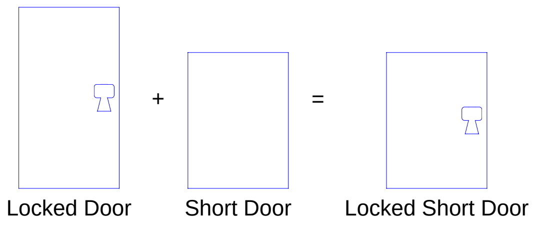
- A ***mixin*** is a class extension without a superclass
- Mixins are more reuseable than class extensions
- Mixins preserve the single-inheritance programming model

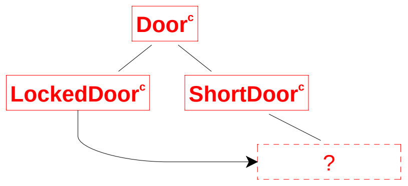
Motivating Example: Door Classes in a Maze Adventure Game



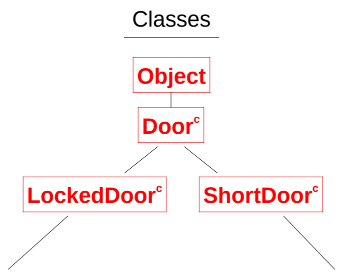


Combining Locked and Short Doors

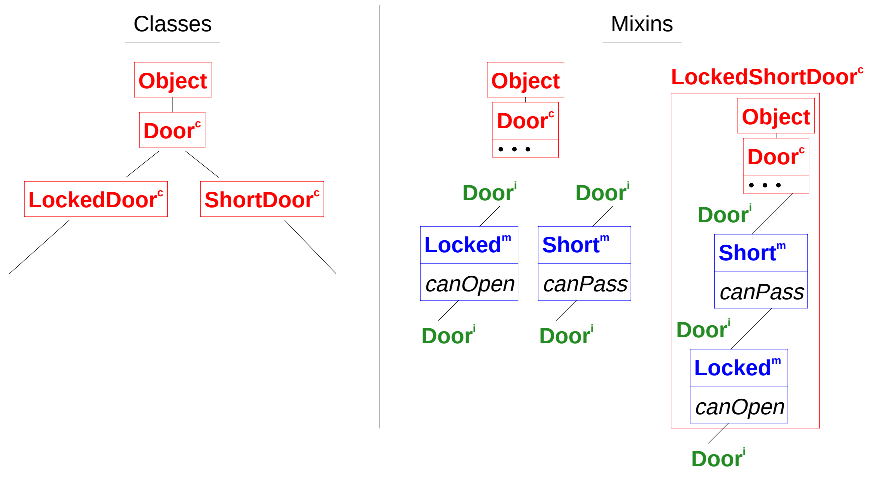




Mixins Allow Combinations



Mixins Allow Combinations



Mixins Allow Combinations

Classes

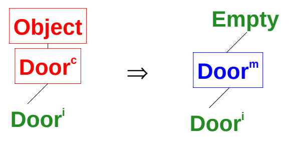
```
class LockedDoor^c extends Door^c {
  boolean canOpen(Person^c p) {
    ....
  }
}

class ShortDoor^c extends Door^c {
  boolean canPass(Person^c p) {
    ....
  }
}

/* LockedShortDoor^c? */
```

Mixins

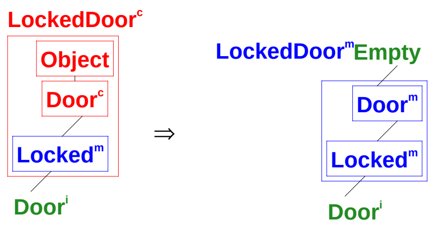
```
mixin Locked^m extends Door^i {
  boolean canOpen(Person^c p) {
    ....
  }
}

mixin Short^m extends Door^i {
  boolean canPass(Person^c p) {
    ....
  }
}

class LockedDoor^c = Locked^m(Door^c);
class ShortDoor^c = Short^m(Door^c);
class LockedShortDoor^c
   = Locked^m(Short^m(Door^c));
```

Mixins Replace Classes

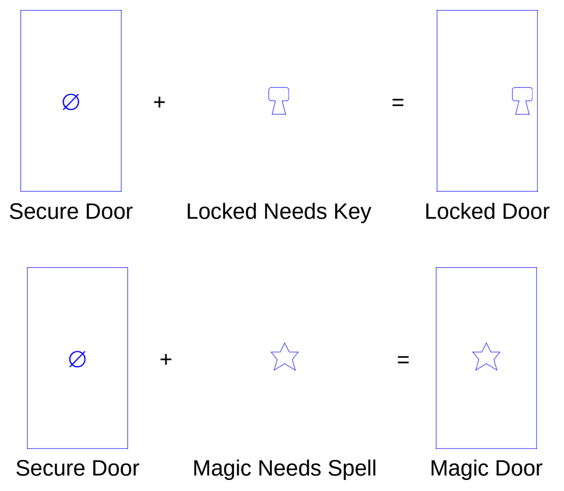
Empty is a special built-in interface



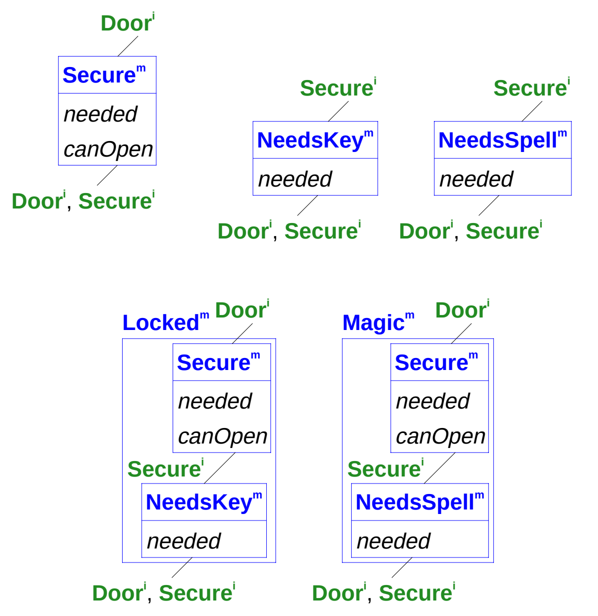
Mixin applications can be replaced with mixin compositions



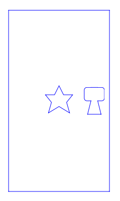
Locked and Magic Doors are Secure Doors



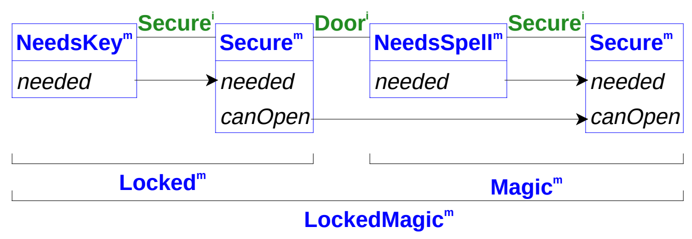
Locked and Magic Door Mixins as Compositions



Locked Magic Doors



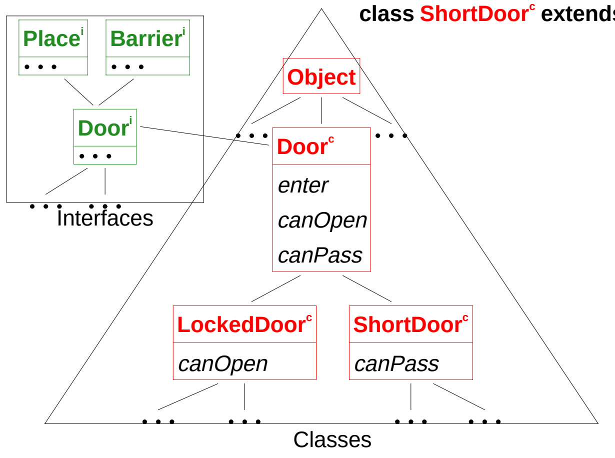
LockedMagic^m = **Locked^m compose Magic^m**



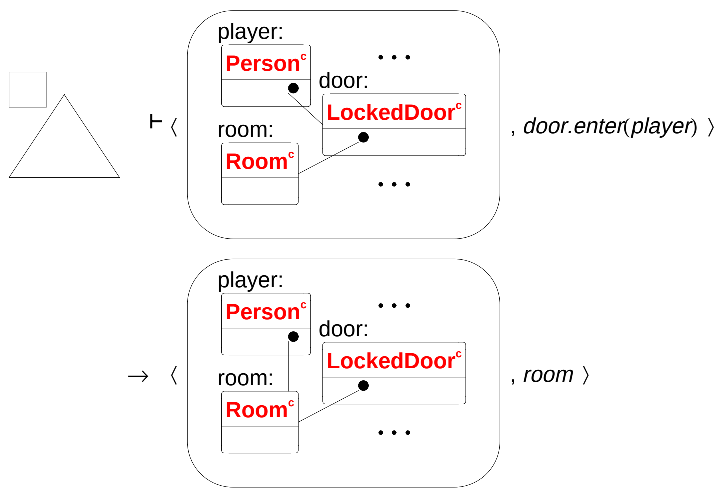
- **Door^i** does not contain *needed*, so there are two distinct *needed* methods in **LockedMagic^m**

Type Checking for Classes

interface Door^i extends Place^i, Barrier^i ....
class Door^c extends Object implements Door^i {
  Room^c *enter*(**Person^c** *p*) { .... }
  boolean *canOpen*(**Person^c** *p*) { .... }
  boolean *canPass*(**Person^c** *p*) { .... } }
class LockedDoor^c extends Door^c ....
class ShortDoor^c extends Door^c ....

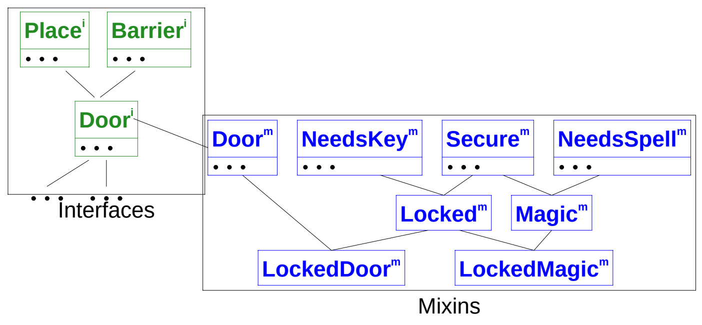


Evaluation for Classes



Type Checking for Mixins

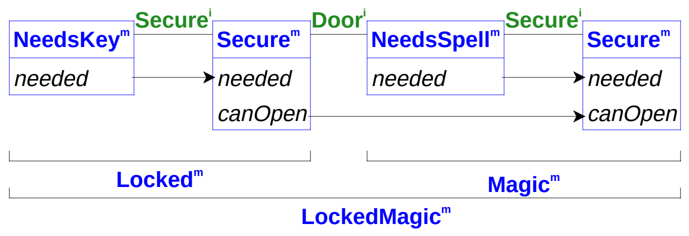
Locked^m = **Secure**^{m} **compose** **NeedsKey**^{m}
Magic^m = **Secure**^{m} **compose** **NeedsSpell**^{m}
. . .



- composite mixin ⇒ linear chain of atomic mixins
- parents = supertypes, parents ≠ subsumable types

“Viewable As” Relation

X subsumes Y ⇔ X is viewable as Y



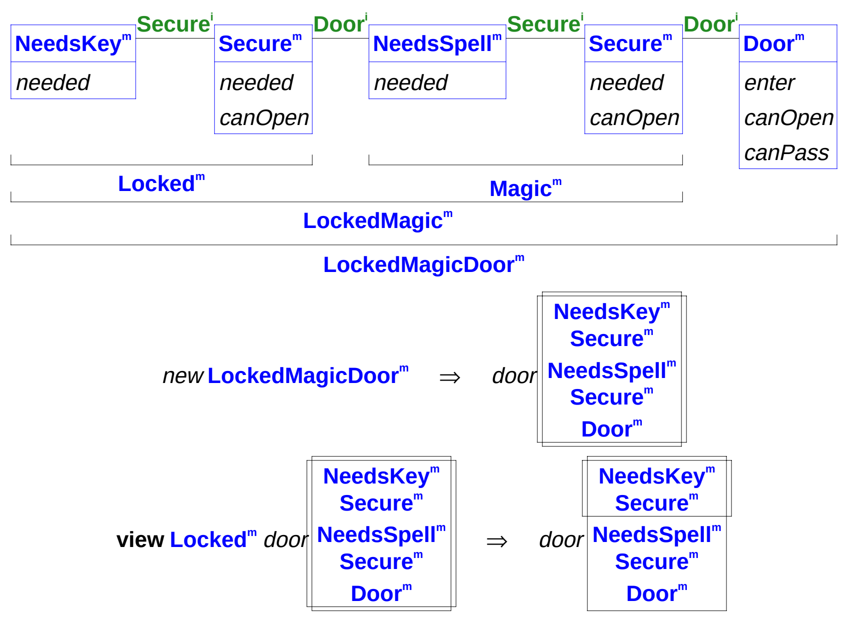
- **LockedMagic**^{m} is viewable as **Locked**^{m} and **Magic**^{m}
- **Locked**^{m} and **Magic**^{m} are viewable as **Secure**^{m}
- **LockedMagic**^{m} is *not* viewable as **Secure**^{m} because **Secure**^{m} is ambiguous in **LockedMagic**^{m}

Mixin Coercions Require Run-time Work

```
Object get(Secure^m o) {
  return o.needed();
}

LockedMagicDoor^m door = new LockedMagicDoor^m;
get(view Locked^m door); /* ==> key */
get(view Magic^m door);  /* ==> magic book */
```

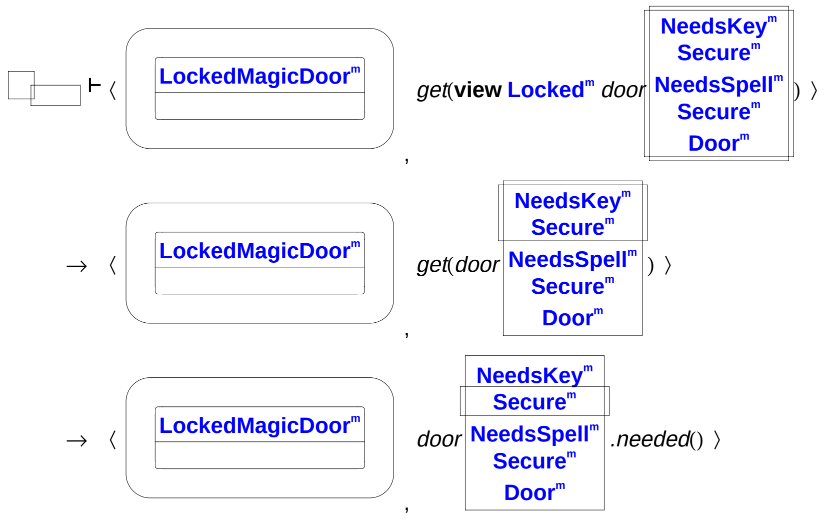
- Intermediate coercions allow *door* as a **Secure**^{m}
- *o.needed*() accesses a different method each time
- Method dispatching depends on the history of run-time coercions

Coercions Recorded with Views



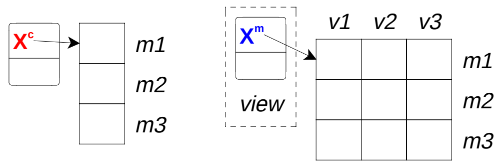
Mixin Evaluation

- Values are object-view pairs
- Coercions adjust the run-time view of an object reference


⊢ ⟨
LockedMagicDoor^m
, get(view Locked^m door) ⟩
→ ⟨
, get(door) ⟩
→ ⟨
, door.needed() ⟩
NeedsKey^m Secure^m NeedsSpell^m Secure^m Door^m

Implementing Mixins

- Every object reference is double-wide: half for object and half for view
- Method lookup requires a two-dimensional virtual table per instantiated chain



- *Cost of mixins = cost of interfaces*
- No cost to programs that do not use mixins

Mixins

- Locally, programming with mixins is the same as single-inheritance classes...
- ... but the programmer is forced to ‘‘program to an interface, not an implementation’’
- Mixin code is more reusable than class code
- Cost of mixins is reasonable (same as interfaces)