**Reusability ≡ Inheritance?**

- Codesharing in Object Oriented Systems is mostly **inheritance-centric**.
- Inheritance itself comes in different flavours:
  - single inheritance
  - multiple inheritance
  - mixin inheritance
- All flavours of inheritance tackle problems of **decomposition** and **composition**

**Duplication**

- Convenient implementation needs **second order types**, only available with
  - Mixins or Templates
**Duplication**

- Convenient implementation needs *second order types*, only available with ~ Mixins or ~ Templates
- With multiple inheritance, read/write Code is essentially *identical but duplicated*

**Lack of Control**

- Common base classes are shared or duplicated at class level
- Linearization overrides all identically named ancestor methods in parallel

---

**Control**

- Common base classes are shared or duplicated at class level
- Linearization overrides all identically named ancestor methods in parallel
- `super` as ancestor reference vs. qualified specification

---

**Control**

- *Fine-grained specification* of duplication or sharing
Inappropriate Hierarchies

- Implemented methods `acquireLock/releaseLock` to high

- Implemented methods `acquireLock/releaseLock` to high

- High up specified methods `turn obsolete`, but there is no statically safe way to remove them

Is Implementation Inheritance even an Anti-Pattern?
Excerpt from the Java 8 API documentation for class Properties:

"Because Properties inherits from HashTable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail..."

⚠️ Misuse of inheritance
Implementation Inheritance itself as a pattern for code reusage is often misused!
→ All that is possible will once be done!

(De-)Composition problems

All the problems of
- Duplication
- Fragility
- Lack of fine-grained control
are centered around the question

“How do I distribute functionality over a hierarchy”

→ functional (de-)composition

The idea behind Traits

- A lot of the problems originate from the coupling of implementation and modelling
  - Interfaces seem to be hierarchical
  - Functionality seems to be modular

⚠️ Central idea
Separate Object creation from modelling hierarchies and assembling functionality.

→ Use interfaces to design hierarchical signature propagation
→ Use traits as modules for assembling functionality
→ Use classes as frames for entities, which can create objects
Classes and Methods – again

The building blocks for classes are
- a countable set of method names \( \mathcal{N} \)
- a countable set of method bodies \( \mathcal{B} \)

Classes map names to elements from the flat lattice \( \mathcal{B} \) (called bindings), consisting of:
- attribute offsets \( \in \mathbb{N}^+ \)
- method bodies \( \in \mathcal{B} \) or classes \( \in \mathcal{C} \)
- \( \bot \) (yet undefined)
- \( \top \) in conflict

and the partial order \( \sqsubseteq \) for each \( m \in \mathcal{B} \)

Definition (Abstract Class \( \in \mathcal{C} \))

A partial function \( c : \mathcal{N} \rightharpoonup \mathcal{B} \) is called abstract class.

Definition (Interface and Class)

An abstract class \( c \) is called
- interface if \( \forall n \in \text{pre}(c) \cdot c(n) = \bot \).
- (concrete) class if \( \forall n \in \text{pre}(c) \cdot \bot \subseteq c(n) \subseteq \top \).

Traits – Composition

Definition (Trait \( \in \mathcal{T} \))

An abstract class \( t \) is called trait iff \( \forall n \in \text{pre}(t) \cdot t(n) \notin \mathbb{N}^+ \)

\( \) is without attributes

The trait sum \( + : \mathcal{T} \times \mathcal{T} \rightharpoonup \mathcal{T} \) is the componentwise least upper bound:

\[
(c_1 + c_2)(n) = b_1 \sqcup b_2 =
\begin{cases}
  b_2 & \text{if } b_1 = \bot \lor n \notin \text{pre}(c_1) \\
  b_1 & \text{if } b_2 = \bot \lor n \notin \text{pre}(c_2) \\
  b_2 & \text{if } b_1 = b_2 \\
  \top & \text{otherwise}
\end{cases}
\]

with \( b_i = c_i(n) \)

Trait-Expressions also comprise:
- exclusion \( : \mathcal{T} \times \mathcal{N} \rightharpoonup \mathcal{T} \):

\[
(t - a)(n) = \begin{cases}
  \text{undf} & \text{if } a = n \\
  t(n) & \text{otherwise}
\end{cases}
\]

- aliasing \( \rightarrow \) \( : \mathcal{T} \times \mathcal{N} \times \mathcal{N} \rightharpoonup \mathcal{T} \):

\[
t[a \rightarrow b](n) = \begin{cases}
  t(n) & \text{if } n \neq a \\
  t(b) & \text{if } n = a
\end{cases}
\]

Traits \( t \) can be connected to classes \( c \) by the asymmetric join:

\[
(c \sqcup t)(n) = \begin{cases}
  c(n) & \text{if } n \notin \text{pre}(c) \\
  t(n) & \text{otherwise}
\end{cases}
\]

Traits – Concepts

Trait composition principles
- Flat ordering All traits have the same precedence under +
- Precedence Under asymmetric join \( \sqcup \), class methods take precedence over trait methods
- Flattening After asymmetric join \( \sqcap \), Non-overridden trait methods have the same semantics as class methods

⚠ Conflicts …

arise if composed traits map methods with identical names to different bodies

Conflict treatment
- Methods can be aliased \( \rightarrow \)
- Methods can be excluded \( \neg \)
- Class methods override trait methods and sort out conflicts \( \sqcup \)

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:
- Definition of curried second order type operators + Linearization

Explicitly: Traits differ from Mixins
- Traits are applied to a class in parallel, Mixins sequentially
- Trait composition is unordered, avoiding linearization effects
- Traits do not contain attributes, avoiding state conflicts
- With traits, glue code is concentrated in single classes
Disambiguation

Traits vs. Mixins vs. Class-Inheritance
All different kinds of type expressions:
- Definition of curried second order type operators + Linearization
- Finegrained flat-ordered composition of modules

Explicitly: Traits differ from Mixins
- Traits are applied to a class in parallel, Mixins sequentially
- Trait composition is unordered, avoiding linearization effects
- Traits do not contain attributes, avoiding state conflicts
- With traits, glue code is concentrated in single classes

Disambiguation

Traits vs. Mixins vs. Class-Inheritance
All different kinds of type expressions:
- Definition of curried second order type operators + Linearization
- Finegrained flat-ordered composition of modules
- Definition of (local) partial order on precedence of types wrt. MRO

Explicitly: Traits differ from Mixins
- Traits are applied to a class in parallel, Mixins sequentially
- Trait composition is unordered, avoiding linearization effects
- Traits do not contain attributes, avoiding state conflicts
- With traits, glue code is concentrated in single classes

Disambiguation

Traits vs. Mixins vs. Class-Inheritance
All different kinds of type expressions:
- Definition of curried second order type operators + Linearization
- Finegrained flat-ordered composition of modules
- Definition of (local) partial order on precedence of types wrt. MRO
- Combination of principles

Explicitly: Traits differ from Mixins
- Traits are applied to a class in parallel, Mixins sequentially
- Trait composition is unordered, avoiding linearization effects
- Traits do not contain attributes, avoiding state conflicts
- With traits, glue code is concentrated in single classes

Traits in the Context of Modularity Problems

Decomposition Problems
- Duplicated Features ... can easily be factored out into unique traits.
- Inappropriate Hierarchies – Trait composition for reusable code concentrates inheritance on shaping interface relations.

Composition Problems
- Conflicting Features – Traits have no state, other conflicts resolved via exclusion, aliasing or overriding.
- Lack of Control – During trait composition precedence is chosen separately for each feature.
- Dispersal of Glue Code ... deferred to and concentrated in the final class.
- Fragile Hierarchies – Trait details are hideable due to missing hierarchy.
Can we augment classical languages by traits?