Inheritance Revisited

- **Motivation: Reuse**
  - Specialization: The class has everything that it’s superclass has plus a couple of more items

  ![Shape](shape.png)
  ![Square](square.png)

- **Alternatives?**
  - Aggregation

  ![Computer](computer.png)
  ![Harddisk](harddisk.png)

Subclassing

- Many languages, as discussed, couple subtyping with inheritance.
  - E.g., when inheriting from a superclass, all methods are inherited.
  - All visible fields of the parent class are visible in the child class.
  - Inheritance forms a structural subtype.

- Subtyping through Inheritance is also called **Subclassing**.

Inheritance without Subtyping

- We mentioned Eiffel.
- Another example is private inheritance in C++
  - public functions of the parent class become private in the child class.

- If subtyping is an “is-a” relationship and aggregation a “has-a”, what would private inheritance be?
  - The designers of C++ say: “implemented in terms of”.
Multiple Inheritance

- Motivation:
  - Why not:

Example C++:

```cpp
class Researcher { /* ... */ };
class Teacher { /* ... */ };
class Professor: public Researcher, public Teacher;
```

Ambiguity

class Researcher {
    int papers;
    public:
        Researcher(int p) { papers = p; }
        int getProductivity() { return papers; }
};
class Teacher {
    int courses;
    public:
        Teacher(int c) { courses = c; }
        int getProductivity() { return courses; }
};
class Professor: public Researcher, public Teacher {
    public:
        Professor(int p, int c): Researcher(p), Teacher(c) { }
};
```

Ambiguity

- Solution: client needs to resolve ambiguity through explicit selection.
  - Breaks abstraction
  - Client needs to know implementation details

Interesting thought: Fragile Base Class Problem and Ambiguity
Ambiguity

- **Merging method**
  - Explicitly resolving the ambiguity in the child class

```cpp
class Professor: public Researcher, public Teacher {
    public:
    Professor(int p, int c) : Researcher(p), Teacher(c) {};
    int getProductivity() {
        return Teacher::getProductivity() + Researcher::getProductivity();
    }
};
```

Ambiguity

- **Renaming**
  - Explicitly disambiguate the different functions in the child class

```cpp
eiffel:
class Professor inherits Teacher
rename getProductivity as getTeaching
redefine getTeaching end
Researcher end
```

Repeated Inheritance

- How many addresses does a Professor instance have?

```
Person
  address: A
  extends
  extends
  Researcher
    papers: int
    extends
    extends
    Professor
  Teacher
    courses: int
    extends
    extends
```

The Diamond Problem

- **C++**

```
Person
  address: A
  extends
  extends
  Researcher
    papers: int
  Teacher
    courses: int
  Professor
```
Virtual Inheritance in C++

- This is the default in Eiffel
- Problem: Who has to call the Person constructor?
  - Initialization problem
  - Researcher and Teachers calls are not performed at runtime
  - Professor as the smallest common subtype must explicitly call it.

Multiple Inheritance

- Multiple Inheritance corresponds to the idea of an object having multiple roles
- Common, e.g., in UIs
- Some aspects of MI are challenging
  - The Diamond Problem
  - Initialization
  - Composition/Modularity
    - E.g., what if the constructor of an intermediate subclass relies on side effects of the superclass constructor
- MI languages typically use some form of linearization.

Avoiding Multiple Inheritance

- Interfaces
  - Declare methods but not provide implementations
  - Classes can implement multiple interfaces but need to implement all of them
  - Merging is explicitly done in the class.
- Interfaces can inherit from other interfaces
- Implementing an Interface forms a subtype

Traits

- Point in the design space between interfaces and classes/multiple inheritance.
- Traits define the signature of methods like interfaces
- Unlike interfaces, traits can declare default implementations for methods.
  - Trait composition is symmetric, results in a flattened view
    - Need to explicitly disambiguate
  - Trait composition is orthogonal to inheritance
Traits in Scala

```scala
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}

class Point(xc: Int, yc: Int) extends Similarity {
  var x: Int = xc
  var y: Int = yc
  def isSimilar(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x == x
}
```

Mixins

- Different form of reuse
- Orthogonal to classes, define methods and states that can be mixed into various existing classes
- Cross-cutting functionality
- Can be considered an increment to the superclass
- Use the (linear) inheritance composition

Mixing-in Traits in Scala

Traits can be used to alter the behavior of the class

```scala
class Mem {
  var value: Int = 0
  def set(v: Int) = { value = v }
  def get: Int = value
}

trait Restore extends Mem {
  var old: Int = 0;
  override def set(v: Int) = {
    old = value;
    super.set(v)
  }
  def restore = { super.set(old) }
}
```

Mixing-in Traits in Scala

```scala
object Main extends App {
  val m = new Mem with Restore;
  m.set(42); println(m.get); m.set(10); println(m.get); m.restore; println(m.get);
}
```
Mixing-in Traits in Scala

Also possible to add the mixin statically

```scala
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}

trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) {
    while (hasNext) f(next)
  }
}
```

```scala
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s.charAt i; i += 1; ch }
}
```

```scala
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

Traits in Scala, Summarized

Unlike interfaces, Traits in Scala can
- Have fields
- Declare new methods
- Override superclass methods

Traits extend (up to) exactly one superclass and may inherit from multiple other traits.

When mixing in traits, class must be subclass of its traits' superclasses

Java 8 Interfaces with Defaults

Interfaces can have default implementations of methods
- Why?

Example

```java
public interface Iterable<T> {
  Iterator<T> iterator();

  default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
      action.accept(t);
    }
  }
}
```