Extension and composition

- Extension implements the *is-a* relation; composition implements the *has-a* relation.

![Diagram showing class relationships with 'is-a' and 'has-a' relations]
Extension and composition (cont.)

```
Player
  <-----
    is-a

PerfectPlayer

Player
  <-----
    has-a

MoveStrategy
```
A *function object*’s only purpose is to provide the functionality implemented by a method.

**Warning:** Saying the purpose of an *object* is to *do* something is often a sign of faulty design. i.e. It probably doesn’t need to be represented as an object, but rather a responsibility of some other more coherent object.
Player

has-a

«interface» MoveStrategy

PerfectStrategy

SimpleStrategy
Inheritance, composition and reuse (cont.)

- The advantages of inheritance are:
  - code reuse
  - polymorphism

- A superclass and subclass are strongly coupled.
  - A change to a superclass specification affects its subclasses and their clients as well as its own clients.
Composition also offers code reuse, since core classes can be used in the construction of a different composed class.

It is possible to change an object’s behavior dynamically at run time.

It supports stronger encapsulation than inheritance.

It is easy to change the specification of the composed class without affecting the core class.
Some of the functionality of a Student is “state dependent.”

Example:

```java
int classification = s.classification();
if (classification == Student.JR_DIVISION){
    //handle JR_DIV case
} else if (classification == Student.UNDERGRAD)
    //handle UNDERGRAD case
} else if (classification == Student.GRADUATE}
    ...
```

A difficulty with subclassing

- The class of an object is fixed when the object is created.
- But, the class of an object may change over time (from Freshman to Sophomore, for example), necessitating the copying of information from the old object to the new object.
- Unfortunately, other elements of the system may still reference the old object.
A better approach

- Use composition rather than employing extension directly.
- Define an interface (or abstract class) that isolates state-dependent behavior.
- Equip an object with a "state-defining" component that implements the interface.
- Different behaviors are achieved by defining different subclasses.
- State transitions become explicit and can be delegated to the state subclasses.
public class Student {
    ...
    public void register () {
        classification.register();
    }
    ...
    private Classification classification;
}
Abstract classes

- Abstract classes are used as a basis from which to build extensions.
- Abstract classes are for structuring the system design rather than for providing run-time objects.
Abstract class guidelines

- An abstract class should be used to model a generalized object, not simply to capture common functionality.
- The decision to generalize must be made with care; any change to an abstract class propagates to its descendants and their clients.
- An abstract class should “factor out” common implementation details of its concrete subclasses.
- Abstraction provides opportunities to exploit polymorphism.
Interface

- An interface in Java is much like an abstract class, but with no constructors, method bodies, or component variables.
- Contains only method specifications (abstract methods) and named constant definitions.
A class implements an interface in much the same way it extends an abstract class.

Example:

```java
public class FlowLayout implements LayoutManager {...}
```
Extending Interfaces

- One interface can extend another interface in much the same way that one class can extend another class.

- Example:

```java
public interface LayoutManager2 extends LayoutManager {
    void addLayoutComponent (Component comp, Object constraints);
    Dimension maximumLayoutSize (Container target);
    float getLayoutAlignmentX (Container target);
    float getLayoutAlignmentY (Container target);
    void invalidateLayout (Container target);
} /* Any class that implements LayoutManager2 must implement these five methods in addition to the five methods specified by LayoutManager. */
```
Multiple inheritance

- For interfaces only, inheritance can extend more than one parent.

```java
public interface DataIO extends DataInput, DataOutput {...}
```
Multiple inheritance (cont.)

```java
public class RandomAccessFile
    implements DataInput, DataOutput { ... }
```
Interfaces vs. abstract classes

- Abstract classes
  - The purpose of a class is to provide an implementation.
  - An abstract class should be used to implement the generalization relationship, factoring out common implementation details from the subclasses.
  - Concrete classes implement specific differences and extend functionality.

- Interfaces
  - An interface defines only a specification.
  - Two classes that implement the same interface are related only in that they support the same abstract functionality defined by the interface.
Interfaces vs. abstract classes

- Abstract classes are used to form hierarchies of implementations.
- Interfaces promote abstraction and reuse across class hierarchies.
- Interfaces specify only functionality and can extend one or more existing interfaces.
Glossary

*code reuse*: use of an existing class definition to construct a new class, either by extending the existing class, or by employing the existing class as a component.

*composed class*: a class constructed by putting together existing classes as components.

*core class*: a class that is used as a component to construct a composed class.

*function class*: a class whose primary purpose is to provide an implementation of some functionality. A function class elevates a method to class status. The class contains the method as its unique public functionality.

*interface*: a language structure that encapsulates a collection of method specifications. An interface defines a restricted view of a collection of mostly unrelated classes, by specifying a set of functionalities all classes in the collection support.

*multiple inheritance*: the ability of a class to have more than one parent. In Java, every class (except *Object*) has a single class parent, but can have multiple interface parents.