Delegation

Delegation is the process of entrusting a power/authority/responsibility to another agent (component, object ...).
In OO software we consider delegation in three forms:

- to a generalization class (inheritance parent),
- to a specialization class (inheritance child), and
- to an unrelated class.

Up-calls: inheritance

Consider two related classes:

```
class A extends C {
  void method() {
    WA; // some A-specific code
    X;
    YA; // more A-specific code
  } ... }
class B extends C {
  void method() {
    WB; // some B-specific code
    X;
    YB; // more B-specific code
  } ... }
```

Inheritance (cont'd)

We can factor the common code into the parent:

```
class C {
  protected final void helperMethod() { X; }
}
class A extends C {
  void method() {
    WA; // some A-specific code
     helperMethod();
    YA; // more A-specific code
  } ... }
class B extends C {
  void method() {
    WB; // some B-specific code
     helperMethod();
    YB; // more B-specific code
  } ... }
```

Inheritance (cont'd)

Sometimes it's appropriate for the common code to be the parent's implementation of the method:

```
class Parent {
  void someMethod() {
    // parent's implementation of someMethod
  }
}
class Child extends Parent {
  void someMethod() {
    super.someMethod(); // call Parent's implementation
    // additional Child-specific implementation
  }
  ...
}
```

Inheritance (cont'd)

For constructors there is special syntax in Java (and C++) for calling the parent's constructor:

```
class Parent {
  Parent( params ) {
    // initialization of Parent's members
  }
}
class Child extends Parent {
  Child( params ) {
    super(args); // call to parent's constructor
                 // this can only be the first
                 // statement in Child's constructor
    // child initialization
  }
}
```

Down-calls: template methods

Suppose the following pattern occurs in the child classes:

```
class Child {
  void someMethod() {
    variable declarations;
    common code part 1;
    child specific code;
    common code part 2;
  }
}
```

We could make two helper methods in the parent
However, if the two common parts share local variables it is quite awkward and requires the child to declare and route the variables.

Template methods (cont'd)

Template pattern solves the problem:

```
class Parent {
  void someMethod() {
    variable declarations;
    common code part 1;
    hookMethod();
    common code part 2;
  }
  void hookMethod() {
    defaultImplementation;
  } ...
}
class Child extends Parent {
  void hookMethod() {
    Child-specific implementation;
  } ...  }
```

Example: Painting in Java Swing

In the library:

```
class JComponent extends java.awt.Container { ...
  public void paint(Graphics g) {
    for each child, c, call c.paint(g)
    paintComponent( g );
  }
  protected void paintComponent( Graphics g ) {
    /* do nothing */ }
  ... }
```

In the client code:

```
class SomeComponent extends JComponent { ...
  protected void paintComponent( Graphics g ) {
    code special to SomeComponent} ...  }
```

Delegation to unrelated class

What if the use of common code appears in unrelated classes?
Delegate the common work to a helper class.
Example: Observer Helper (Manager) . . .

Library class (common code)

```
public class ObserverHelper {
  private List listOfObservers = new ArrayList();
  public void notify() {
    Iterator it = listOfObservers.iterator();
    while(it.hasNext()) {
      Observer observer = (Observer)(next());
      observer.update();
    }
  }
  public void addObserver(Observer observer) {
    listOfObservers.add(observer);
  }
  public void removeObserver(Observer observer) {
    listOfObservers.remove(observer);
  }
}
```

Client code

```
class ConcreteSubject extends Something {
  private ObserverHelper helper = new ObserverHelper();
  private notify() { helper.notify(); }
  public addObserver(Observer obs) {
    helper.addObserver( obs ); }
  public removeObserver(Observer obs) {
    helper.addObserver( obs ); }
...}
```

Compare this to GoF Observer, which uses inheritance.

Varying the helper: Strategy Pattern

Inheritance can't be changed at runtime, but associations can. The concrete class of a helper object can be determined at construction (or later) time.

Strategy Pattern

```
class ConfigurableClass {
  private HelperInterface helper;
  // Constructor
  public ConfigurableClass (HelperInterface helper) {
    ...
    this.helper = helper;
    ...
  }
  public changeHelper(HelperInterface helper) {
    this.helper = helper;
  }
  void someMethod( ) {
    ...
    helper.someHelperMethod();
    ...
  } ... }
```

Example: Layout Managers in AWT

In AWT each Container object has a LayoutManager object.

- LayoutManager is an interface:

```
void addLayoutComponent(String name, Component comp)
void layoutContainer(Container parent);
Dimension minimumLayoutSize(Container parent);
Dimension preferredLayoutSize(Container parent);
void removeLayoutComponent(Component comp);
```

- Containers use their layout object to determine where to place their Components.
- Mix and match: library LayoutManagers can be used in custom Containers or vice-versa.

Layout Examples

FlowLayout	puts component in horizontal lines (like words in a paragraph).
GridLayout	puts components in a grid.
GridBagLayout	a more flexible grid.
BorderLayout	puts one component in the middle and others around the edges.