Polymorphism and Software Reuse
(Budd chapters 14 - 17)

Plan

- Overloading
- Overriding
- The Polymorphic Variable
- Generics

Definition of Polymorphic

- **Polymorphous**: Having, or assuming, various forms, characters, or styles.
- From greek routes, **Poly** = many, and **Morphos** = form
- In programming languages, used for a variety of different mechanisms.
- Usage of the term is confused by the fact that it means something slightly different in the functional programming community than it does in the OO world.

- One name, many meanings.

Major Forms of Polymorphism in Object Oriented Languages

- There are four major forms of polymorphism in object-oriented languages:
  - **Overloading** (ad hoc polymorphism) -- one name that refers to two or more different implementations.
  - **Overriding** (inclusion polymorphism) -- A child class redefining a method inherited from a parent class.
  - **The Polymorphic Variable** (assignment polymorphism) -- A variable that can hold different types of values during the course of execution. It is called **Pure Polymorphism** when a polymorphic variable is used as a parameter.
  - **Generics** (or Templates) -- A way of creating general tools or classes by parameterizing on types.

Two Approaches to Software Reuse

- Many polymorphisms, one goal: software reuse
- One of the major goals of OOP is software reuse. Two different approaches to reuse:
  - **Inheritance** -- the is-a relationship.
  - **Composition** -- the has-a relationship.

Advantages and Disadvantages of Each Mechanism

- Composition is **simpler**, and clearly indicates what operations are provided.
- Inheritance makes for **shorter code**, possibly increased functionality, but makes it more difficult to understand what behavior is being provided.
- Inheritance may open to the door for unintended usage, by means of unintended inheritance of behavior.
- Easier to change underlying details using composition (i.e., change the data representation).
- Inheritance may permit polymorphism
- **Understandability**: each has different complexity issues (size versus inheritance tree depth)
Overloading

- Overloading based on scopes
- Overloading based on type signatures
- Coercion, Conversion and Casts
- Redefinition

A Definition of Overloading

- We say a term is *overloaded* if it has two or more meanings.
- Most words in natural languages are overloaded, and confusion is resolved by means of context.
- Same is true of OO languages. There are two important classes of context that are used to resolve overloaded names
  - Overloading based on Scopes
  - Overloading based on Type Signatures

Overloading Based on Scopes

- A *name scope* defines the portion of a program in which a name can be used, or the way it can be used.
- An advantage of scopes is that the same name can appear in two or more scopes with no *ambiguity*.
- E.g. methods with same name in different classes.

Resolving Overloaded Names

- This type of overloading is resolved by looking at the *type (class)* of the *receiver*.
- Allows the same name to be used in unrelated classes.
- Since names need not be distinct, allows short, easy to remember, meaningful names.

Overloading Based on Type Signatures

- A different type of overloading allows multiple implementations in the same scope to be resolved using *type signatures*.
  
  ```java
  class Example {
  // same name, three different methods
  int sum (int a) { return a; }
  int sum (int a, int b) { return a + b; }
  int sum (int a, int b, int c) { return a + b + c; }
  }
  ```

- A *type signature* is the combination of argument types and return type. By looking at the signature of a call, can tell which version is intended.
- Note that *resolution* is almost always performed at *compile* time, based on static types, and not dynamic values.

Conversion and Coercions

- *Coercion* and *conversions* represent a change of type.
  - *Coercion* is an implicit change of type; e.g. `int i; double x; x = i + x;` // integer will be converted to real
  - *Conversion* usually represents an explicit change in type.
  - Often, operator used is called a *cast*, e.g. `x = ( (double) i) + x;`
- Change representation/pointer
- When one adds coercion/conversions, resolving overloaded method calls can get very complex.
**Redefinitions**

- A *redefinition* occurs when a child class changes the type signature of a method in the parent class.

Two different types of rules are used to resolve name:

- The *merge* model.
  - The scope of the child is merged with the scope of the parent.
- The *hierarchical* model.
  - Scopes are separate.
  - Each scope is examined in turn.
  - When a scope containing the name is found, then the best match in that scope will be selected.

**Example Illustrating Redefinition Models**

- The following example will illustrate the difference in these two models.

  ```java
  class Parent {
    public void example (int a) {
      System.out.println("in parent method");
    }
  }
  
  class Child extends Parent {
    public void example (int a, int b) {
      System.out.println("in child method");
    }
  }
  
  Child aChild = new Child();
  aChild.example(3);
  ```

  - Will execute parent method in Java and C# (Merge model) and give error in C++ (Hierarchical model).
  - Delphi allows programmer control over which model to use.

**Overriding**

- A method in a child class *overrides* a method in the parent class if it has the same name and type signature.

**Difference from Overloading**

- Like overloading, there are two distinct methods with the same name. But there are differences:
  - Overriding only occurs in the context of the parent/child relationship
  - The type signatures must match in overriding
  - Overriding is resolved at run-time, not at compile time.

**Notating Overriding**

- In some languages (Smalltalk, Java) overiding occurs automatically when a child class redefines a method with the same name and type signature.
- In some languages (C++) overiding will only occur if the parent class has declared the method in some special way (example, keyword `virtual`).
- In some languages (Object Pascal) overiding will only occur if the child class declare the method in some special way (example, keyword `override`).
- In some languages (C#, Delphi) overiding will only occur if both the parent and the child class declare the method in some special way.

  ```java
  class Parent { // C# example
    public virtual int example (int a) { ... }
  }
  
  class Child : Parent {
    public override int example (int a) { ... }
  }
  ```

**Replacement and Refinement**

- There are actually two different ways that overriding can be handled:
  - A *replacement* totally and completely replaces the code in the parent class with the code in the child class.
  - A *refinement* executes the code in the parent class, and adds to it the code in the child class.

- Most languages use both types of semantics in different situations. Constructors, for example, almost always use refinement.
Reasons to use Replacement

• There are a number of reasons to use replacement of methods.
  – The method in the parent class is abstract, it must be replaced.
  – The method in the parent class is a default method, not appropriate for all situations.
  – The method in the parent can be more efficiently executed in the child.

Downside of Replacement

• The downside of replacement semantics is that there is no guarantee that the child class will have any meaning at all similar to the parent class.
• For example, a child class could redefine sqrt to compute the cube root of its argument.
• This goes back to the difference between subclasses and subtypes.
• A refinement makes this more difficult to do, since whatever the parent does is guaranteed to be part of the child.
• This is why most languages use refinement semantics for constructors.
• This guarantees that whatever initialization the parent class performs will always be included as part of the initialization of the child class.

Overriding versus Shadowing

• It is common in programming languages for one declaration of a variable to shadow a previous variable of the same name:

```java
class Silly {
  private int x; // an instance variable named x
  public void example (int x) { // x shadows instance var
    int a = x+1;
    while (a > 3) {
      int x = 1; // local variable shadows parameter
      a = a - x;
    }
  }
}
```

• Shadowing can be resolved at compile time, does not require any run-time search.

Covariance and Contravariance

• Frequently it seems like it would be nice if when a method is overridden we could change the argument types or return types.
• A change that moves down the inheritance hierarchy, making it more specific, is said to be covariant.
• A change that moves up the inheritance hierarchy is said to be contravariant.

```java
class Parent {
  void test (covar : Mammal, contravar : Mammal) {
  }
}
class Child extends Parent {
  void test (covar : Cat, contravar : Animal) {
  }
}
```

• While appealing, this idea runs into trouble with the principle of substitution.

Summary (overriding)

• An override occurs when a method in the child classes uses the same name and type signature as a method in the parent class.
• Unlike overloading, overriding is resolved at run-time.
• There are two possible means for an overriding, replacement and refinement.
• A name can shadow another name. Shadowing is resolved at compile time.
• A change in the type signature can be covariant or contravariant, if it moves down or up the type hierarchy.

The Polymorphic Variable

• A polymorphic variable is a variable that can hold values of different types during the course of execution.
Simple Polymorphic Variables

- We saw simple polymorphic variables in the employee example (section Java: polymorphism)

```java
public class PayrollSystemTest {
  public static void main( String[] args ) {
    DecimalFormat twoDigits = new DecimalFormat( "0.00" );
    // create Employee array
    Employee employees[] = new Employee[4];
    // initialize array with Employees
    employees[0] = new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
    employees[1] = new CommissionEmployee( "Sue", "Jones", "222-22-2222", 10000, .06 );
    employees[2] = new BasePlusCommissionEmployee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
    for ( int i = 0; i < employees.length; ++i ) {
      output += "earned $" + employees[ i ].earnings() + "\n";
    }
  }
}
```

The Receiver Variable

- The most common polymorphic variable is the one that holds the receiver during the execution of a method.
- Major role of `this` in C++ and Java is to access a data field or serve as receiver when methods are passed to “oneself”.
- When an overridden method is executed, it is the method in the child class that is executed, not the method in the parent class.
- Consider a class `Window` with subclasses `TextWindow` and `GraphicsWindow`.

Downcast (Reverse Polymorphism)

- It is sometimes necessary to undo the assignment to a polymorphic variable.
- That is, to determine the variables true dynamic value, and assign it to a variable of the appropriate type.
- This process is termed downcasting, or, since it is undoing the polymorphic assignment, reverse polymorphism.
- Various different syntaxes are used for this.
  ```java
  Parent aVariable = ...;
  Child aChild;
  if (aVariable instanceof Child) {
    aChild = (Child) aVariable;
  }
  ```

Pure Polymorphism

- A polymorphic method (also called pure polymorphism) occurs when a polymorphic variable is used as an argument.
- Different effects are formed by using different types of value.
  ```java
class StringBuffer {
  String append (Object value)
  { return append(value.toString()); }
  ...}
```
- Different objects implement `toString` differently, so the effects will vary depending upon the argument.

Summary (polymorphic variable)

- A polymorphic variable is a variable that can reference more than one type of object
- Polymorphic variables derive their power from interaction with inheritance, and overriding.
- A common polymorphic variable is the implicit variable that maintains the receiver during the execution of a method
- Downcasting is the undoing of a polymorphic assignment
- Pure polymorphism occurs when a polymorphic variable is used as an argument.