Polymorphism

n. the condition of occurring in several different forms

via subtyping and dynamic binding

Welcome to a mini-course on programming language semantics and type systems.
The language features you have learned so far in 111 and 230 are useful, but inheritance opened the door for us to discuss some more interesting programming language concepts over the next week or two. If you enjoy these ideas, take 251 in the spring!

What is a programming language?

Syntax:
How is a program expressed?
What are the syntactic building blocks of a program?
What combinations of symbols constitute valid programs?

Semantics
What does a program mean?
Abstractly, how is a program evaluated (run) and what is the result?

Type system (not strictly necessary, but highly useful)
What types of data and operations are present in programs?
Will a program ever apply operations to incompatible data?

Type System
Every value has a type describing what it is.
7 : int "ABC" : String new Cow() : Cow true : boolean 4.8 : float
(pronounce: "has type" or "is of type")

Every variable has a type describing what it holds.
int x; String abc; Cow c; boolean b; float f;

Every method has a type describing what it takes and returns when called.
int abs(int x) String substring(int i) void moo() float milk(boolean m)

Every operation has rules describing relationships between its components.
x = x * 7 "ABC".substring(0) : abc(x * 7) new Cow().milk(f)

Type checker: checks if type system rules are followed.
static (compile-time) vs. dynamic (run-time) vs. hybrid

Subtyping S <: T
Type compatibility relation.
If S is a subtype of T (S <: T), then values of type S:

- have all the visible features of values of type T.
  Subclass implies subtype in Java*
  class Square extends Rectangle { ... }
  Square <: Rectangle

- may be treated as values of type T.
  Subsumption: if S <: T and x : S then x : T

Reflexive (T <: T) and transitive (R <: S and S <: T implies R <: T)

*But not vice versa, and not in all languages.
Subtype Polymorphism

- Java references are subtype polymorphic: a reference of type T may refer to values of type T or any subtype of T.
- Rectangle myShape = new Rectangle(3.0, 4.0);
- myShape = new Square(4.0);
- public void draw(Rectangle r) { ... }
  ...  
  draw(new Square(9.0));

Polymorphism via dynamic dispatch

public static double addAreas(Rectangle r, Rectangle s) {
  return r.area() + s.area();
}

public class Rectangle {
  public void printArea() {
    System.out.println(this.area());
  }
  ...
}

Static vs. Dynamic Binding (compile-time) (run-time)

Binding = attaching a thing and a name.

myShape.area();

When is the identifier area() bound to a particular method implementation?

static/early binding occurs statically, at compile time:
Method is selected based on the declared type of the reference.

dynamic/late binding (a.k.a. dynamic dispatch) occurs dynamically at run time, just before the method is invoked:
Method is selected based on the object to which the reference refers.

Why is this flexible?

Example: Pay various types of employees with one polymorphic method.
StaffMember.java
abstract public class StaffMember {
  protected String name;
  protected String address;
  protected String phone;
  // Constructor: Sets up this staff member using the specified info.
  public StaffMember (String eName, String eAddress, String ePhone) {
    name = eName;
    address = eAddress;
    phone = ePhone;
  }
  public abstract double pay();
}

Volunteer.java
//********************************************************************
// Volunteer.java Java Foundations
// Represents a staff member that works as a volunteer.
//********************************************************************
public class Volunteer extends StaffMember {
  // Constructor: Sets up this volunteer using the specified information.
  public Volunteer (String eName, String eAddress, String ePhone) {
    super (eName, eAddress, ePhone);
  }
  // Returns a zero pay value for this volunteer.
  public double pay() {
    return 0.0;
  }
}

Employee.java
//********************************************************************
// Employee.java Java Foundations
// Represents a general paid employee.
//********************************************************************
public class Employee extends StaffMember {
  protected String socialSecurityNumber;
  protected double payRate;
  // Constructor: Sets up this employee with specified information.
  public Employee (String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
    ...}
  // Returns the pay rate for this employee.
  public double pay() {
    return payRate;
  }
}

Executive.java
//********************************************************************
// Executive.java Java Foundations
// Represents an executive staff member, who can earn a bonus.
//********************************************************************
public class Executive extends Employee {
  private double bonus;
  public Executive (String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
    ...}
  // Computes and returns the pay for an executive, which is the
  // regular employee payment plus a one-time bonus.
  public double pay() {
    double payment = super.pay() + bonus;
    bonus = 0; // already paid
    return payment;
  }
}
Hourly.java

// Hourly.java Java Foundations
// Represents an employee that gets paid by the hour.
// ********************************************************************
public class Hourly extends Employee {
    private int hoursWorked;
    // Constructor: Sets up an hourly employee using specified info.
    public Hourly(String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
        ...
    }
    // Computes and returns the pay for this hourly employee.
    public double pay() {
        double payment = payRate * hoursWorked;
        hoursWorked = 0; // already paid
        return payment;
    }
}

Firm.java

// Firm.java Java Foundations
// Demonstrates polymorphism via inheritance.
// ********************************************************************
public class Firm {
    public static void main(String[] args) {
        Staff personnel = new Staff();
        personnel.payday();
    }
}

Staff.java

// Staff.java Java Foundations
// Represents the personnel staff of a particular business.
// ********************************************************************
public class Staff {
    private StaffMember[] staffList;
    // Constructor: Sets up the list of staff members.
    public Staff () {
        staffList = new StaffMember[5];
        staffList[0] = new Executive("Tonya", "123 Main Line", 
                                  "555-0469", "678-5845", 2423.07);
        staffList[1] = new Employee("Paulie", "456 Off Line", 
                                   "555-8190", "321-0987", 2345.76);
                                   "555-9876", "876-5432", 3210.09);
        staffList[3] = new Hourly("Michael", "678 Fifth Ave.", 
                                "555-8698", "543-2109", 10.55);
        ((Executive)staffList[0]).awardBonus(500.00);
        ((Hourly)staffList[3]).addHours(40);
    }
    // Pays all staff members.
    public void payday () {
        double amount;
        for (int count=0; count < staffList.length; count++) {
            amount = staffList[count].pay(); // polymorphic
            if (amount == 0.0) {
                System.out.println("Thanks!");
            } else {
                System.out.println("To Be Paid: "+ amount);
            }
        }
    }
}

Staff.java continued

// Staff.java Java Foundations
// Pays all staff members.
// ********************************************************************
public void payday () {
    double amount;
    for (int count=0; count < staffList.length; count++) {
        amount = staffList[count].pay(); // polymorphic
        if (amount == 0.0) {
            System.out.println("Thanks!");
        } else {
            System.out.println("To Be Paid: "+ amount);
        }
    }
}
**Interfaces: explicit subtype polymorphism**

- Formal CS111 "contract"
- A Java interface is composed of abstract methods (and constants)
- Establishes a type

```java
public interface Doable {
    public void doThis();
    public int doThat(int num);
    public boolean doTheOther();
}
```

All methods in an interface are implicitly abstract.

None of the methods in an interface are given a definition (body).

A semicolon immediately follows each method signature.

**Encryptable.java**

// Encryptable.java Java Foundations

// Represents the interface for an object that can be encrypted and decrypted.

```java
public interface Encryptable {
    public void encrypt();
    public String decrypt();
}
```

**A class implements an interface by using the implements keyword providing implementations for each abstract method in the interface.**

**WHY?:**

An interface cannot be instantiated.

Interface methods default to public.

An implementing class:
- must define all the interface methods.
- may implement other methods.
- establishes a type that is a subtype of the interface type. CanDo <: Doable
SecretTest.java

// SecretTest.java Java Foundations
// Demonstrates the use of a formal interface.
//
public class SecretTest {
    // Creates a Secret object and exercises its encryption.
    public static void main (String[] args) {
        Secret hush = new Secret("Wil Wheaton is my hero!");
        hush.encrypt();
        System.out.println(hush); // prints %mjfyts%n%r&mjwt&
        hush.decrypt();
        System.out.println(hush); // prints Wil Wheaton is my hero!
    }
}

Secret.java

// Secret.java Java Foundations
// Represents a secret message that can be encrypted and decrypted.
//
import java.util.Random;
public class Secret implements Encryptable {
    private String message;
    private boolean encrypted;
    private int shift;
    private Random generator;
    //-----------------------------------------------------------------
    // Constructor: Stores the original message and establishes
    // a value for the encryption shift.
    //-----------------------------------------------------------------
    public Secret (String msg) {
        message = msg;
        encrypted = false;
        generator = new Random();
        shift = generator.nextInt(10) + 5;
    }
    (more…)
    //-----------------------------------------------------------------
    // Encrypts this secret using a Caesar cipher. Has no effect if
    // this secret is already encrypted.
    //-----------------------------------------------------------------
    public void encrypt () {
        if (!encrypted) {
            String masked = "";
            for (int index=0; index < message.length(); index++) {
                masked = masked + (char)(message.charAt(index) + shift);
            }
            message = masked;
            encrypted = true;
        }
    }
    public String decrypt() {
        if (encrypted) {
            String unmasked = "";
            for (int index=0; index < message.length(); index++) {
                unmasked = unmasked + (char)(message.charAt(index) - shift);
            }
            message = unmasked;
            encrypted = false;
        }
        return message;
    }
    (etc., methods …)

Multiple Interfaces

◆ A class can implement multiple interfaces
  ∗ Multiple subtyping relations across inheritance hierarchies

class Horse implements Mammal, Vehicle {
    // all methods of both interfaces
}

9.3

Why do friends of Artie/Cowboys breathe more easily vs. Multiple inheritance?
Can we reuse this code to sort other types of arrays?

```java
public static void sort (String[] a) {
    String max;    // maximum String so far
    int maxIndex;  // index of maximum String
    int i, j;
    for (j = a.length - 1; j > 0; j--) {
        maxIndex = 0;
        max = a[0];
        for (i = 1; i <= j; i++) {
            if (a[i].compareTo(max) > 0) {
                max = a[i];
                maxIndex = i;
            }
        }
        swap(a, maxIndex, j);
    }
}
```

**Why is this powerful? Java Interfaces**

- The Java standard class library has many useful interfaces

```java
public interface Comparable {
    public int compareTo(Comparable other);
}
```

(Real version is slightly better – we’ll see soon.)

- The String class implements Comparable, giving us the ability to put strings in lexicographic order

**The Comparable Interface**

- Any class can implement Comparable to provide a mechanism for comparing objects of its type.
  ```java
  if (obj1.compareTo(obj2) < 0)
      System.out.println("obj1 is less than obj2");
  ```
  - It’s up to the programmer to determine what makes one object < than another
  - I.e., you may define the compareTo method of an Employee class to order employees by name (alphabetically) or by employee number
  - The implementation of the method can be as straightforward or as complex as needed for the situation
The Iterator Interface

- An iterator is an object that provides a means of processing a collection of objects, one at a time.

\[\text{java.util} \]

Interface \texttt{Iterator<E>}

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hasNext()</strong></td>
</tr>
<tr>
<td><strong>next()</strong></td>
</tr>
<tr>
<td><strong>remove()</strong></td>
</tr>
</tbody>
</table>

- Pair with \texttt{java.lang.Iterable<T>} for use in for-each loops.
  - \texttt{<T> iterator()} for more on \texttt{<T> soon...}

Other interfaces you know...

```java
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}

public abstract class AbstractButton extends JComponent {
    private ActionListener[] listeners = new ActionListener[...];
    private int numListeners = 0;
    ...
    public void addActionListener(ActionListener al) {
        ...
        ... // resize if needed
        listeners[numListeners++] = al;
    }
    protected void fireActionPerformed(ActionEvent e) {
        for (int i = 0; i < numListeners; i++) {
            listeners[i].actionPerformed(e);
        }
    }
}
```

Explicit subtyping relation

Java subtyping is established only via explicit declaration, with \texttt{extends} or \texttt{implements}.

```java
class Artist {
    public void draw() {
        ...
    }
}

interface Lottery {
    public void draw();
}
```

```java
Lottery x = new Artist();
```