Object-Oriented Programming

CS 99 – Summer 2000

Michael Clarkson

Lecture 9

Administration

- Prelim 2 graded
- Lab 8 due now
- Lab 9 posted ?

Agenda

- OOP
  - Evolution
  - Three principles
- Basic OOP in Java

Evolution of OOP

- Functions
- Modules
- Abstract Data Types
- Classes and Objects

Structured Analysis & Design

- Invented 1970s
- Coincided with elimination of GOTO
- Identify functions
  - Group code for repeated tasks into one place
  - One programmer can write a function that many programmers can use without knowing implementation details
- Problem: only local and global scope
  - Names become a problem

Modular Programming

- Module: abstract mechanism for managing names
- Public and private namespaces
  - Public is the interface provided to users
  - Private is the implementation used in the module
- “Need to know” philosophy
  - Users of module should know only enough to use module
  - Programmers of module should know only enough to write it
- Problem: only one module (e.g., one Car) can be present in a program at a time

Abstract Data Types

- Programmer-defined data type
- Set of values and operations on those values
- Allows:
  - Extension of language with new types
  - Hiding of implementation details
  - Creation of multiple instances of type
- Problem: still not good enough for managing complexity of really large programs

OOP

- Idea: a program is a collection of cooperating objects sending messages to one another
- Grew out of simulation techniques from the 1960s
- Adds innovations over ADTs that give it extra power:
  - Message passing – emphasis on data, not function
  - Polymorphism – interpretation of message can vary depending on what object receives it
  - Relationships between objects
  - Behavior and Rules

Fundamental Concept

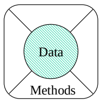
- The Object
  - Software package with:
    - Attributes (data)
    - Methods (code) that act on data
  - Data is not accessible to users of object
  - Access to data granted through methods
  - Self-governing

Principles of OOP

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

- Object contains (encapsulates) all its own code and data



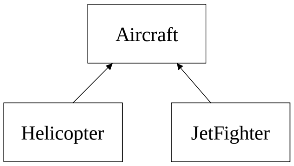
- Information hiding: other objects don't know how an object manages its data
  - Don't have access to either the data or the code
- Objects interact through well-defined messages

Inheritance

- Aircraft has:
  - Manufacturer, ID#, weight, cost, etc.
  - Take off, land, turn, etc.
- Can refine for more specific aircraft:
  - Helicopter: has propellers, can hover
  - Jet fighter: has missiles, can fire them

Inheritance [2]

- Generalization/Specialization: is-a relationship



Inheritance [3]

- Abstraction mechanism for sharing similarities among classes while preserving differences
- Superclass (parent) is refined into a subclass; subclass inherits from superclass
- Subclass inherits attributes and methods from parent
- Subclass adds its own attributes, methods, possibly replaces those of parent
- **Allows code to be reused**

Polymorphism

- Messages can be interpreted differently based on the receiving object
- Subclass replaces a parent's method with its own
  - e.g.: takeOff() different for Helicopter than Aircraft
- But if subclass doesn't replace, parent's method is used
  - e.g. JetFighter uses Aircraft's takeOff()

Basic OOP in Java

- Overloading
- Subclasses
- Interfaces

Method Signature

- Includes
  - Name of method
  - Number of parameters
  - Types of parameters
  - Order of parameters
- For example, `main(String[])` is the signature for `main`
- Does not include return type

Overloading

- Overloaded methods are one type of polymorphism in Java
  - Purists: not actually polymorphism
- Overloaded methods are methods with the same name but different signatures
  - Example: multiple constructors
- Java selects which method to call based on the signature

`println`

- Has 10 overloaded versions:
  - `println()`
  - `println(boolean)`
  - `println(char)`
  - `println(char[])`
  - `println(double)`
  - `println(float)`
  - `println(int)`
  - `println(long)`
  - `println(Object)`
  - `println(String)`

`println` [2]

- When program is compiled, compiler determines types of arguments and then *binds* the call to the correct version of `println`
- This allows one method name to exhibit several types of behavior, thus polymorphism
- Convenience – we only have to remember one method name!

Overloaded Constructors

- Again, convenience
- Allows multiple ways to create an object
- Programmer can choose the most suitable

Overloaded Operators

- Operators can also be overloaded
- Plus sign:
  - `int + int`
  - `double + double`
  - `String + String`
- Java doesn't allow programmers to overload operators
  - Some languages do
  - `Complex + Complex      // C++`
  - `Complex.plus(Complex)  // Java`

Subclasses

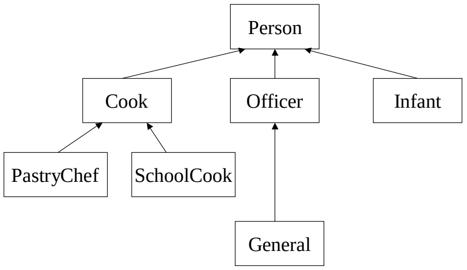
- Java supports inheritance through the use of subclasses
- New subclasses are derived from existing classes (superclasses)
- Subclasses inherit the methods and attributes of all their parents
  - Subject to visibility rules

Subclasses [2]

- Subclasses are created with the `extends` keyword:

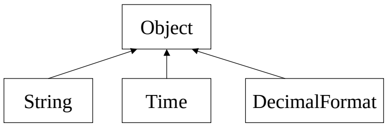
```
class Person {
      ...
}
class Cook extends Person {
      ...
}
class PastryChef extends Cook {
      ...
}
```

Class Hierarchy



Java Class Hierarchy

- The parent class of all classes in Java is Object
- All classes are subclasses of Object



`protected`

- Another visibility modifier
- Similar to `private`, but subclasses can see the member

	`public`	`protected`	`private`
Same class	✓	✓	✓
Subclass	✓	✓	
Unrelated class	✓		

Inheritance

```
class Box {
   protected double width, height, depth;
   public Box(double w, double h, double d) {
        width = w; height = h; depth = d;
   }
   public double volume() {
        return width * height * depth;
   }
}
class WeightedBox extends Box {
   protected double weight;
   public WeightedBox(double w, double h, double d, double m){
        width = w; height = h; depth = d; weight = m;
   }
   public getWeight() { return weight; }
}
```

Inheritance [2]

- WeightedBox inherited the fields and methods of its superclass
- Can access them as if they were its own members:

```
WeightedBox w = new WeightedBox(10, 20, 15, 34.3);
System.out.println(“Volume = ” + w.volume()); // 3000.0
System.out.println(“Weight = ” + w.getWeight()); // 34.3
```

Inheritance [3]

- Subclasses can *override* inherited methods and replace them with their own code (polymorphism)

```
class InsulatedBox extends Box {
   public volume() {
        return width * height * depth * .75;
   }
}
```

Interfaces

- Abstraction of interactions with an object
- Set of public methods that describes services provided by an object
- Says nothing about how services are provided (implementation)
- Says what a class must do, but nothing about how it does it

Interfaces [2]

- Conceptually similar to roles that people play
- For example, I provide these interfaces:
  - Grader
  - Instructor
  - PetOwner
- Rick also provides the Grader interface
- Objects can provide several different interfaces, and you won't always know (or need to know) what all of them are

Java Interfaces

- Syntactically similar to classes:

```
public interface Calculator {
  Number add(Number n1, Number n2);
  Number subtract(Number n1, Number n2);
  Number multiply(Number n1, Number n2);
  Number divide(Number n1, Number n2);
  Number sqrt(Number n);
}
```

Java Interfaces [2]

- Full syntax:

```
public interface name {
  return-type method-name1(parameter-list);
  return-type method-name2(parameter-list);
  ...
  type final-varname1 = value;
  type final-varname2 = value;
  ...
}
```

Java Interfaces [3]

- If an interface is declared as `public`:
  - Methods are automatically `public`
  - Fields are automatically `public final static`
- Multiple classes can implement an interface:

```
public interface SquareRootCalculator {
  double sqrt(double num);
}
```

Implementing Interfaces

```
class NewtonRaphson implements SquareRootCalculator {
  double sqrt(double num) {
     // N-R method code
  }
}
class EasyWay implements SquareRootCalculator {
  double sqrt(double num) {
     return Math.sqrt(num);
  }
}
```