Principles of Computer Science

Inheritance

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

A Triangle class in Java

```
class Triangle {
  public float base;
  public float height;

  public Triangle(float a, float b) {
    this.base = a;
    this.height = b;
  }
  float area() {
    return base * height / 2;
  }
  void enlarge(float factor) {
    this.base = Math.sqrt(factor) * this.base;
    this.height = Math.sqrt(factor) * this.height;
  }
}
```

Members

Methods
- area
- enlarge

Inheritance

- Consider the class **ColoredTriangle**
  - Members:
    - `float base;`
    - `float height;`
    - `Color color;`
  - Methods:
    - `float area()`
    - `void enlarge()`
    - `void setColor(Color newColor)`
- This is nearly identical to Triangle, but with more members and methods.
- Golden software design principle: Reuse existing code

Deriving Subclasses

- In Java, we use the reserved word extends to establish an inheritance relationship

```
class Car extends Vehicle
{
   // class contents
}
```

Reusing Code

```
class ColoredTriangle extends Triangle {
  public Color color;
  public void changeColor(Color newColor) {
    this.color = newColor;
  }
}
```

- **ColoredTriangle** is a subclass of **Triangle**
- Inheritance in Java
  – There can be at most ONE superclass (unlike C++)

Constructor

- Subclasses often require their own constructors which will require the constructor of their super class.
- Declare new constructors in the subclass as normal.
- In the subclass constructor, one can call super(...) to refer to the constructor of the super class.

Example:

```
class ColoredTriangle extends Triangle {
  public Color color;
  public ColoredTriangle(Color color) {
    super(1.0f, 1.0f);
    this.color = color;
  }
  public void changeColor(Color newColor) {
    this.color = newColor;
  }
}
```

Visibility of members and methods:

- Members and methods can be very sensitive and should, often, be protected.
- In object oriented programming, one can use **modifiers** to change the visibility and accessibility of the members and methods.
- Different visibility modifieres:
  - public
  - private
  - protected

Example: inappropriate visibity

```
class Professor {
  public String name;
  private float salary;
  private String office;
  public Professor(String name, float startSalary) {
    ...
  }
  public void moveOffice(String newOffice) {
    ...
  }
  public void promotion() {
    ...
  }
}
```

Example: inappropriate visibity

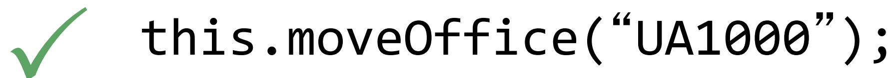
```
class SomeBody {
...
Professor smartGuy = new Professor(...);
...
✓ System.out.println(smartGuy.name);

✗ System.out.println(smartGuy.salary);

✓ smartGuy.name = “Spam it.”

✓ smartGuy.moveOffice(...);

✓ smartGuy.promotion();
...

}
```

Example: appropriate visibity

```
class Professor {
  private String name;
  private float salary;
  private String office;
  public Professor(String name, float startSalary) {
    ...
  }
  protected void moveOffice(String newOffice) {
    ...
  }
  protected void promotion() {
    ...
  }
  public String getName() { return this.name; }
  public String getOffice() { return this.office; }
  protected String setOffice(String newLocation) {
   …
  }
}
```

Example: appropriate visibity

```
class Supervisor extends Professor {
  ...
```




```
}
```

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier abstract, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interface in Java

- Interface is declared using the interface keyword.
- Interfaces can **have methods only,** not members.
- Methods can be modified by visibility modifier
- One interface can extend another interface using the extends keyword:
  - interface Square extends Rectangle {
    ...
    }

Example

- Consider an interface **AreaShape** which describes shapes which have areas.

```
interface AreaShape {
  public float area();
}
```

```
interface AreaShape {
  abstract public float area();
}
```

Implementations of interfaces

- A class implements an interface if it has **all** the methods listed in the interface.
- A class can implement many interfaces.
- A class must **declare itself** to be an implementation of an interface.

Implementation of interface in Java

- The keyword is **implements**.
- General syntax:

```
class class-name implements interface1, interface2, ...
```

- Consider another interface: **ShapeShifter**

```
interface ShapeShifter {
  public enlarge(float factor);
}
```

Implementation of interfaces in Java

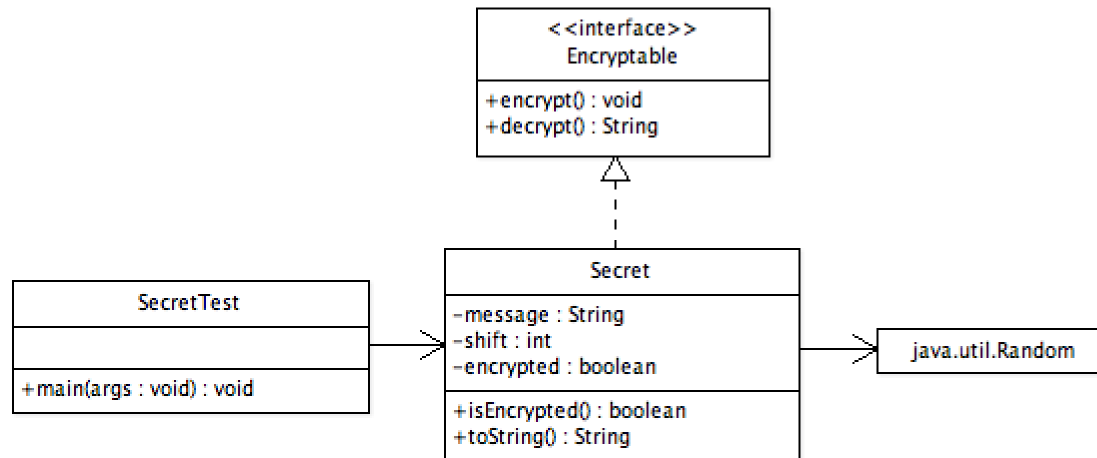
```
class Triangle implements AreaShape, ShapeShifter {
 ...
 public float area() {
   return base * height / 2;
 }
 public void enlarge(float factor) {
   base *= Math.sqrt(factor);
   height *= Math.sqrt(factor);
 }
}
```

Why interfaces?

- Interfaces and their implementations are written separately (possibly by different developers).
- At the interface development phase, developer focuses on the **functionality** of the class.
- At the implementation development phase, developer focuses on the technical details of how to realize the functionalities in the interface.
- Interfaces ensure that nothing is left out.
- Interfaces can be used to achieve polymorphism.
  [more on this]

Interfaces

- An example of an interface in use



Abstract classes

- Interfaces have limitations
  - No methods can be implemented.
  - There can be **no** members.
- What if:
  - Some methods can already be implemented at the "interface" level while some methods are left **abstract**?
  - The "interface" is to have some members?
- Solution: **Abstract classes**

Abstract classes

- Abstract classes are an immediate stage between classes and interfaces:
  - it can have abstract methods – just like interfaces
  - it can have members and fully implemented methods – just like classes
  - it can have constructors – but see below.
- Abstract classes **cannot** be instantiated into objected directly because it has unimplemented (or abstract) methods.
- Constructors of abstract classes are used only by their sub-classes through the usage of **super(...).**

Abstract Classes

- We use the modifier abstract on the class header to declare a class as abstract:

```
public abstract class Product
{
   // contents
}
```

- Unlike an interface, the abstract modifier must be applied to each abstract method

Example

- Consider a class: **ColoredAreaShape**.
- Member:
  - Color mycolor;
- Method:
  - float area();
  - boolean isLarge();
    This method return true if and only if the area is greater than 100.

Example

```
abstract class ColoredAreaShape {
  protected Color myColor;
  protected ColoredAreaShape() {
    this.myColor = Color.BLACK;
  }
  abstract public float area();
  public boolean isLarge() {
    return (this.area() > 100);
  }
  public Color color() {
    return this.myColor;
  }
}
```

Example

```
abstract class ColoredAreaShape {
  protected Color myColor;
  protected ColoredAreaShape() {
    this.myColor = Color.BLACK;
  }
  abstract public float area();
  public boolean isLarge() {
    return (this.area() > 100);
  }
  public Color color() {
    return this.
  }
}
```

```
class ColoredTriangle extends ColoredAreaShape {
  float base, height;
  public ColorTriangle(...) {
    super();
  }
  public float area() {
    return (base * height) / 2;
  }
}
```

Example

```
abstract class ColoredAreaShape {
  protected Color myColor;
  protected ColoredAreaShape() {
    this.myColor = Color.BLACK;
  }
  abstract public float area();
  public boolean isLarge() {
    return (this.area() > 100);
  }
  public Color color() {
    return this.
  }
}
```

```
class ColoredTriangle extends ColoredAreaShape {
  public float base, height;
  public ColorTriangle(...) {
    super();
  }
  public float area() {
    return (base * height) / 2;
  }
}
```

Example

- ColoredTriangle is a class because it implements all the abstract methods in ColoredAreaShape.
- Consider the following statements:
  - ✓ ColoredTriangle x1 = new ColoredTriangle(...);
  - ✗ ColoredAreaShape x2 = new ColoredAreaShape(...);
  - ✓ ColoredAreaShape x3 = new ColoredTriangle(...);
  - ✓ x1.isLarge()
  - ✓ x1.area()
  - ✓ x3.isLarge()
  - ✓ x1.base
  - ✗ x3.base

More on inheritance

- Consider:

```
class C₂ extends C₁ {
  ...
}
```

- If C_1 is abstract, then C_2 must implement **all** abstract methods of C_1 because C_2 is declared as a class.

- Now consider:

```
abstract class C₂ extends C₁ {
  ...
}
```

- C_1 **must** be abstract. C_2 does not need to implement all the abstract methods.

Method overriding

- Subclasses can optionally [see condition below] redefine the behaviour of methods inherited from the super-class. This is called **method overrideing**.
- The super-class can prevent overriding of certain methods by declaring them as **final**.

Restricting Inheritance

- If the final modifier is applied to a method, then that method cannot be overridden in any descendent classes
- If the final modifier is applied to an entire class, then that class cannot be used to derive any children at all
  - Thus, an abstract class cannot be declared as final
- These are key design decisions, establishing that a method or class should be used as is

Example

```
class Person {
  protected int age;
  protected float income;
  public Person(int age, float income) {
    this.age = age;
    this.income = income;
  }
  public boolean isSenior() {
    return (age >= 65);
  }
  final public void payTax() {
    this.income *= 0.7;
  }
}
```

Example

```
class Person {
  protected int age;
  protected float income;
  public Person(int age, float in
    this.age = age;
    this.income = income;
  }
  public boolean isSenior() {
    return (age >= 65);
  }
  final public void payTax() {
    this.income *= 0.7;
  }
}
```

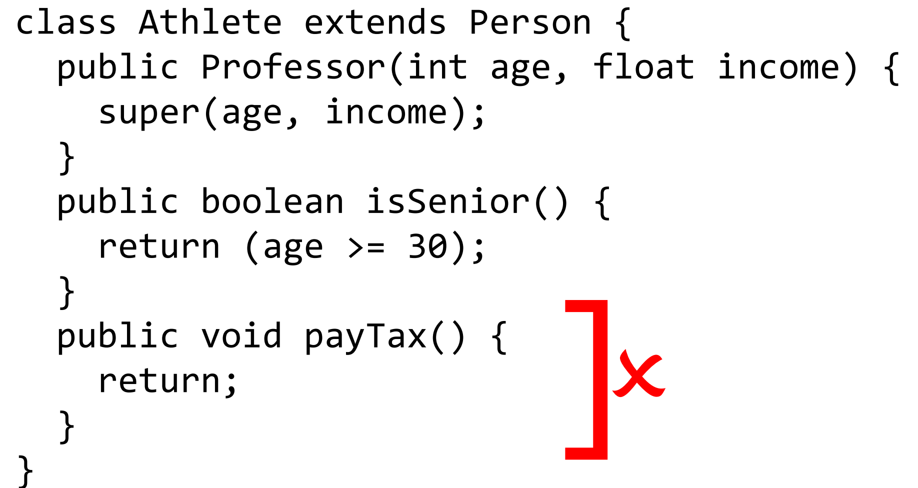
```
class Professor extends Person {
  public Professor(int age, float income) {
    super(age, income);
  }
  public boolean isSenior() {
    return (age >= 85);
  }
}
```

```
class Athlete extends Person {
  public Professor(int age, float income) {
    super(age, income);
  }
  public boolean isSenior() {
    return (age >= 30);
  }
}
```

Example

```
class Person {
  protected int age;
  protected float income;
  public Person(int age, float in
    this.age = age;
    this.income = income;
  }
  public boolean isSenior() {
    return (age >= 65);
  }
  final public void payTax() {
    this.income *= 0.7;
  }
}
```

```
class Professor extends Person {
  public Professor(int age, float income) {
    super(age, income);
  }
  public boolean isSenior() {
    return (age >= 85);
  }
}
```



Polymorphism

- **poly**, as in “polygon”, “polytechnique” etc, means “many”.
- **morphism** means “form”.
- Polymorphism means that one variable refer to objects of different forms.
- Polymorphism is the antonym of types.
  - *int x;* means that *x* must be an integer and nother else.
  - *ColoredAreaShape x;* means x can refer to an object of any class **as long as** the class is a sub-class of *ColoredAreaShape*.

Possible polymorphisms in Objected oriented programming

- Polymorphism does not have any special syntax to learn. It just means that you can use one type of variable to refer to many different types of objects.
- Given `C1 obj=new C1(…);`
- The assignment `C2 x=obj;` is legal if:
  - C1 is a sub-class of C2
  - C1 is an implementation of the interface C2.

READING MATERIAL

- Text:
  - Chapter 8
  - Chapter 9
- Online:
  - http://java.sun.com/docs/books/tutorial/java/index.html
    - Interfaces and inheritance

Summary

- Abstract methods and abstract classes are defined with the ***abstract*** key word
- Abstract methods have no body, and their header must end with a semicolon.
- An abstract method must be overridden in a subclass.
- When a class contains an abstract class, it cannot be instantiated. It must serve as a superclass.