cisc3120
design and implementation of software applications I
spring 2015
lecture # I.5

topics:

- introduction to java, part 5
  - code documentation with `javadoc`
  - inheritance
  - overriding methods
  - polymorphism
  - 'super' keyword
  - 'final' keyword

code documentation with javadoc

- `javadoc` tool uses the special comments enclosed within `/** ... */` and generates html documenation, in the same format as Java API docs
- regular comments in the code with `//` and `/* ... */` are ignored
- javadoc can be used to describe classes, constructors, methods, fields and interfaces
- usage: `$> javadoc [options] [files]`
- some useful options:
  `-d <path>` : saves the doc files in `path`
  `-author` : compiles the docs with the author information (if provided)
  `-private` : includes `private` fields and methods (default is `public` and `protected`)
  `-classpath <path>` : searches for references in `path`
- example:
  `$> javadoc -d docs *.java`
  generates documentation for all `.java` files in the current directory and saves them in `docs` directory

code documentation with javadoc - comments and common tags

- descriptions have to be placed within `/** ... */` immediately before the class, method, field, etc.
- `javadoc` recognizes certain keywords that follows the `@` character, referred to as *tags*
- some common tags:
  - `@author [name]`
  - `@param [param_name] [parameter description]`
  - `@return [description of returned data]`
  - `@throws [exception name] [exception description]`
- to view the documentation, look for the `index.html` file within the documentation `path` if specified by `-d` option

inheritance in java (1)

- classes by themselves are only serve to organize code. the real power of OOP lies in how these classes are related to one another through inheritance.
- *inheritance* is the means by which classes are created out of existing classes
- the idea is to re-use existing code and create *specialized* versions of existing classes when necessary
- in java a class can inherit from another class using the keyword `extends`, example:

```
class Animal {
  public float weight;
  public void eat() {...}
}

class Mammal extends Animal {
  public void breathe() {...}
}
```

inheritance in java (2)

- for the previous example `Animal` is said to be the *superclass* of `Mammal` and `Mammal` is said to be the *subclass* of `Animal`
- class `Mammal` inherits the field `weight` and method `eat()` therefore the following statements would be legal:

```
Mammal human = new Mammal();
human.weight = 80.0;          // inherited from Animal
human.eat();                  // inherited from Animal
human.breathe();              // specific to Mammal
```

- prefixes *super* and *sub* comes from set theory, to represent the relationship between the instances e.g. all Mammal instances are Animals
- this is known as the *is-a* relationship between a subclass and the superclass
- superclasses are also called *base* class or *parent* class
- subclasses are also called *derived* class or *child* class

inheritance in java (3)

- think of the most primitive Java class, `Object` as being at the root of the inheritance tree
- all other classes are "children" or *subclasses* of that class
- here is an example of the inheritance tree for `Integer`:

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.lang.Integer
```

- `Integer` is a subclass of `Number` and `Number` is a subclass of `Object`
- `Integer` is also a subclass of `Object`
- conversely a parent is also called a *superclass*
- `Object` is a superclass of `Number` and `Number` is a superclass of `Integer`
- `Object` is also a superclass of `Integer`

inheritance in java (4)

- as you move DOWN the inheritance tree from the root to the leaf, you are *extending* subclasses from parent classes
- as you move UP the inheritance tree from the leaf to the root, you can say that each subclass instance is-a *more specific* version of its parent
- A subclass **does not** have access to the **private** members of its parent class:

```
class Animal {
  private float weight;
  public void setWeight(float w) { weight = w; }
}

class Mammal extends Animal {
  public gainWeight() {
    weight += 10;          // error!
    setWeight(50);         // okay!
  }
}
```


- If the superclass has `public` or `protected` methods for accessing its private fields, these can also be used by the subclass.
- The `protected` modifier specifies that the member can only be accessed:
  - within its own package
  - by a subclass of its class in another package

Polymorphism

- you can assign an instance of a subclass to a superclass typed reference:

```
Animal a = new Mammal();
```

- but cannot use the superclass reference to call subclass methods unless it is casted as a subtype reference :

```
a.breathe();            // compile-time error.
Mammal m = (Mammal) a;
m.breathe();            // okay!
```

- in the above example, object `m` is treated both as a `Mammal` and as an `Animal` instance, this is called **Polymorphism**, object having multiple forms (from greek words 'poly' and 'morph')

overriding methods (1)

- when you *extend* a class, you can provide your own definition of methods that are derived from the parent class, this is called **overriding**
- example:

```
class Animal {
  public void reproduce() {}
}

class Mammal extends Animal {
  public void reproduce() { // sexual reproduction }
}
```


overriding methods (2)

- a method overriding a superclass method has the same exact signature as the superclass method
- one of the common mistakes when overriding a method is to accidentally implement the method with a different signature (e.g. double parameter instead of a float)
- to prevent this mistake annotation `@Override` is usually added before the method definition, to let the compiler check if the implementation actually overrides a superclass method, example:

```
@Override
public String toString() {...}
```


dynamic binding and hiding

- because the objects are polymorphic, the references may point to an instance of the class itself or an instance of one of its subclasses (e.g. `Animal a` may refer to a an `Animal` object or a `Mammal` object)
- if there are methods overriden by the subclass, the actual selection of the method to call is done in runtime, this is called **dynamic binding**. example:

```
Animal animals[] = new Animal[2];
animals[0] = new Animal();
animals[1] = new Mammal();

animals[0].reproduce();  // calls reproduce() in Animals
animals[1].reproduce();  // calls reproduce() in Mammal
```

- a class method (static method) with the same signature as a class method in the superclass, the method in the subclass `hides` the one in the superclass.
- within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.

overriding methods (3)

```
public class Coin {
    public static String classDescription() {
        return "Coin class";
    }

    public String toString() {
        return "Value:" + getValue();
    }
}

public class Quarter extends Coin {
    public static String classDescription() {
        return "Quarter class";
    }

    @Override
    public String toString() { return "Quarter"; }

    public static void main(String[] args) {
        Quarter quarter = new Quarter();
        Coin coin = quarter;
        System.out.println(coin.classDescription());
        System.out.println(quarter.classDescription());
        System.out.println(coin); // Overriden method
    }
}
```


super keyword

- If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`.

```
public class Quarter extends Coin {
  @Override
  public String toString() {
    String coinDescription = super.toString();
    return "Quarter: "+ coinDescription;
  }
}
```


super() constructor call

- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass - using `super` keyword.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the default constructor of the superclass.
- Invocation of a superclass constructor must be the first line in the subclass constructor.

```
public class Quarter extends Coin {
  public Quarter() {
    value = 25;
    flip();
  }
```

OR

```
  ...
  public Quarter() {
    super(25);
  }
}
```


inheritance example - Employee, Manager

```
class Employee {
    public Employee(String n, double s){
        name = n;
        salary = s;
    }

    public String getName(){ return name; }

    public double getSalary() { return salary; }

    public void raiseSalary(double byPercent){
        double raise = salary * byPercent / 100 ;
        salary += raise ;
    }

    private String name;
    private double salary;
}
```

inheritance example - Employee, Manager (2)

```
class Manager extends Employee {
    public Manager(String n, double s){
        super(n, s);
        bonus = 0;
    }

    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

    public void setBonus(double b){
        bonus = b;
    }

    private double bonus;
}
```


inheritance example - Employee, Manager (3)

```
public class ManagerTest{
  public static void main(String[] args){
    Manager boss = new Manager("Harry", 80000);
    boss.setBonus(5000);

    Employee[] staff = new Employee[3];

    staff[0] = boss;
    staff[1] = new Employee("Tom", 50000);
    staff[2] = new Employee("Jack", 60000);

    for( Employee e : staff )
      System.out.println("name:" + e.getName() + ", salary:" + e.getSalary());
  }
}
```


final classes and methods

- you can declare some or all of a class's methods `final`.
- you use the `final` keyword in a method declaration to indicate that the method cannot be overridden by subclasses.

```
public class Coin {
    final int getDefaultFace() {
        return Coin.HEADS;
    }
}
```

- if you declare an entire class final this will prevent the class from being subclassed