Classes, Objects, Inheritance

Access Modifiers & Packages

- **Access modifiers:**
  - **public:** Can be accessed / invoked by anybody
  - **private:** Can only be accessed / invoked from within same class
  - **protected:** Can only be accessed / invoked from within same class and its subclasses
  - **<no modifier>:** Can be accessed / invoked from within same package

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclasses</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>no modifier</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Packages:**
  - Encapsulate a set of classes and interfaces
  - Hierarchical organization
  - Declaration: `package myfirstpackage;`
  - Examples: `java.math, de.tum.wzw`
Classes, Objects, Inheritance

- Designated class-method main with fixed signature
  
  ```java
  public static void main(String[] args)
  ```
  
is called once at program start

- "Computer has to know where to start":

  ```java
  class BicycleDemo {
      public static void main(String[] args) {
          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
      }
  }
  ```

Overloading

- **Overloading:** Methods with same name but different parameters (types)

  ```java
  class OverloadingDemoClass {
      public int doSomething() {
          return 1 + 1;
      }

      public int doSomething(int param) {
          return param + 2;
      }
  }

  public static void main(String[] args) {
      OverloadingDemoClass odc = new OverloadingDemoClass();
      int result1 = odc.doSomething();
      int result2 = odc.doSomething(33);
  }
  ```

- Method signature comprised of *name* and *parameter types*

Overriding, Hiding

- **Overriding methods**
  - Why? Let subclasses provide a more specialized version of an instance-method
  - How? Subclass defines an instance-method with same signature (name plus number and types of parameters) as defined by super-class

  ```java
  class Bicycle {
      public void speedUp(int increment) {
          speed = speed + increment;
          System.out.println("superclass instance-method");
      }
  }

  class MountainBike extends Bicycle {
      public void speedUp(int increment) {
          speed = speed + 2 * increment;
          System.out.println("subclass instance-method");
      }
  }

  MountainBike mountainBike = new MountainBike();
  mountainBike.speedUp(10);
  ```

  output will be: subclass instance-method
Overriding, Hiding

- Hiding methods
  - Why?
  - Let subclasses provide a more specialized version of a class-method
  - How?
    Subclass defines a class-method with same signature (name plus number and types of parameters) as defined by superclass

```java
class Bicycle {
    public static void myClassMethod(int someInt) { // superclass class-method
        System.out.println("superclass class-method");
    }
    public void myInstanceMethod(int someInt) { // superclass instance-method
        System.out.println("superclass instance-method");
    }
}

class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) { // subclass class-method
        System.out.println("subclass class-method");
    }
    public void myInstanceMethod(int someInt) { // subclass instance-method
        System.out.println("subclass instance-method");
    }
}

Bicycle myBicycle = new Bicycle();
Bicycle myMountainBike = new MountainBike();

myBicycle.myClassMethod(10); // "superclass class-method"
myBicycle.myInstanceMethod(10); // "superclass instance-method"
myMountainBike.myClassMethod(10); // "subclass class-method"
myMountainBike.myInstanceMethod(10); // "subclass instance-method"
```

Polymorphism

- **Polymorphism**: subclass objects may be assigned to superclass variables
  - MountainBike mountainBike = new MountainBike();
  - Bicycle bicycle = mountainBike;
  - **Essential feature** of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the *overridden* ("right") methods are called

```java
Bicycle bike = new MountainBike();
bike.gear = 3; // Ok, gear defined in class Bicycle
bike.speedUp(10); // Overridden method in subclass MountainBike is used
bike.seatHeight = 20; // ERROR! seatHeight is not a field in class Bicycle
```
Polymorphism

- **Polymorphism:** subclass objects may be assigned to superclass variables

```java
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```java
Bicycle bike = new MountainBike();
bike.gear = 3;  // Ok, gear defined in class Bicycle
bike.speedUp(10);  // Overridden method in subclass MountainBike is used
bike.seatHeight = 20;  // ERROR! seatHeight is not a field in class Bicycle
```

---

Polymorphism

- **Polymorphism:** subclass objects may be assigned to superclass variables

```java
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```java
Bicycle bike = new MountainBike();
bike.gear = 3;  // Ok, gear defined in class Bicycle
bike.speedUp(10);  // Overridden method in subclass MountainBike is used
bike.seatHeight = 20;  // ERROR! seatHeight is not a field in class Bicycle
```

---

Polymorphism

- **Purpose of polymorphism:**
  General superclass state and behaviour may be used on all subclass objects

→ Good software design

- Somewhat similar:
  Interfaces provide a blueprint of blueprints, and may be used as type in variable declarations.

Different classes may implement the same interface.

→ The methods which are guaranteed by the interface may be called on objects of all corresponding classes
4 Recursion

Deepening readings:

http://en.wikipedia.org/wiki/Recursion
http://en.wikipedia.org/wiki/Factorial
package flowers;

public class Flower {
    private double amountOfPetals;
    public Flower(double initialAmountOfPetals) {
        if (initialAmountOfPetals > 0) {
            amountOfPetals = initialAmountOfPetals;
        } else {
            System.err.println("Initial amount of petals must be > 0");
        }
    }
    public Flower() {
        amountOfPetals = 200.0 * Math.random();
    }
}

package flowers;

public class Flower {
    private double amountOfPetals;
    public Flower(double initialAmountOfPetals) {
        if (initialAmountOfPetals > 0) {
            amountOfPetals = initialAmountOfPetals;
        } else {
            System.err.println("Initial amount of petals must be > 0");
        }
    }
    public Flower() {
        amountOfPetals = 200.0 * Math.random();
    }
}

package OADemo;

public static void main(String[] args) {
    Flower lily = new Flower(100.0);
    Flower buttercup = new Flower(50.0);
    System.out.println("Lily: ");
    System.out.println("Amount of petals: ", lily.getAmountOfPetals());
    System.out.println("Birch: ");
    System.out.println("Amount of petals: ", buttercup.getAmountOfPetals());
}

package OADemo;

public static void main(String[] args) {
    Flower lily = new Flower(100.0);
    Flower buttercup = new Flower(50.0);
    System.out.println("Lily: ");
    System.out.println("Amount of petals: ", lily.getAmountOfPetals());
    System.out.println("Birch: ");
    System.out.println("Amount of petals: ", buttercup.getAmountOfPetals());
}
4 Recursion

Deepening readings:

**Recursion:** Divide a given problem into subproblems of the same type

- One or more base cases
- Rules to reduce other cases towards base case

**Example:**
- Factorial: 
  \[ n! = \begin{cases} 
  1 & \text{if } n = 0, \\
  (n - 1)! \cdot n & \text{if } n > 0. 
\end{cases} \]
- Fibonacci: 
  \[ f(n) = \begin{cases} 
  1 & \text{if } n = 1, \\
  1 & \text{if } n = 2, \\
  f(n-1) + f(n-2) & \text{if } n > 2. 
\end{cases} \]

**Example:**
- People are standing in queue
- Doorman wants to know how many people are waiting
- What are the base- and general cases?
Recursion

Recursive method calls & Stack

• Local variables and parameters stored on stack
• For each function call, a corresponding stack frame is created

```java
long factorial(int n) {
    long temp;
    if (n == 1) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```java
public static void main(String[] args) {
    long result;
    result = factorial(4);
    System.out.println(result);
}
```
4 Recursion

- **Recursion**: Divide a given problem into subproblems of the same type
  - One or more base cases
  - Rules to reduce other cases towards base case

**Example:**

- **Factorial**
  \[ n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases} \]

- **Fibonacci**
  \[ f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ f(n-1) + f(n-2) & \text{if } n > 2. \end{cases} \]