Introduction To Object-Oriented Programming

Part II: You will learn the difference between functions and methods, how to define accessor (get) methods and mutator (set) methods, how to overload methods such as constructors and why this is regarded as good style.

James Tam

Terminology: Methods Vs. Functions

- Both include defining a block of code that be invoked via the name of the method or function (e.g., `print()`)
- **Methods** a block of code that is *defined within a class definition* (Java example):

```
public class Person
{
    public Person() { ... }

    public void sayAge() { ... }
}
```

- Every object that is an instance of this class (e.g., `jim` is an instance of a `Person`) will be able to invoke these methods.

```
Person jim = new Person();
jim.sayAge();
```

James Tam

Terminology: Methods Vs. Functions (2)

- **Functions** a block of code that is *defined outside or independent of a class* (Python example – it's largely not possible to do this in Java):

```
# Defining method sayBye()
class Person:
    def sayBye(self):
        print("Hosta lavista!")

# Methods are called via an object
jim = Person()
jim.sayBye()

# Defining function: sayBye()
def sayBye():
    print("Hosta lavista!")

# Functions are called without creating an object
sayBye()
```

James Tam

Methods Vs. Functions: Summary & Recap

Methods

- The Object-Oriented approach to program decomposition.
- Break the program down into classes.
- Each class will have a number of methods.
- Methods are invoked/called through an instance of a class (an object).

Functions

- The procedural (procedure = function) approach to program decomposition.
- Break the program down into functions.
- Functions can be invoked or called without creating any objects.

James Tam

Second Example: Second Look

Calls in Driver.java

```
Person jim = new Person();

jim.sayAge();
```

Person.java

```
public class Person {
    private int age;

    public Person() {
        age = in.nextInt();
    }

    public void sayAge() {
        System.out.println("My age
                    is " + age);
    }
}
```

More is needed:
•What if the attribute 'age' needs to be modified later?
•How can age be accessed but not just via a `print()`?

James Tam

Viewing And Modifying Attributes

1) **New terms: Accessor methods**: '`get()`' method
   - Used to determine the current value of an attribute
   - Example:

```
public int getAge()
{
    return(age);
}
```

2) **New terms: Mutator methods**: '`set()`' method
   - Used to change an attribute (set it to a new value)
   - Example:

```
public void setAge(int anAge)
{
    age = anAge;
}
```

James Tam

Version 2 Of The Second (Real) O-O Example

Name of the folder containing the complete example:
`third_accesorsMutators`

James Tam

Class Person

- Notable differences: the constructor is redesigned, `getAge()` replaces `sayAge()`, `setAge()` method added

```
//First version
public class Person
{
    private int age;
    public Person() {
        …
        age = in.nextInt();
    }

    public void sayAge() {
        System.out.println("My age
           is " + age);
    }
}
```

```
//New version
public class Person
{
    private int age;
    public Person() {
        age = 0;
    }
    public int getAge() {
        return(age);
    }

    public void setAge
       (int anAge){
        age = anAge;
    }
}
```

James Tam

Class Driver

```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person();
        System.out.println(jim.getAge());
        jim.setAge(21);
        System.out.println(jim.getAge());
    }
}
```

```
0
21
```

James Tam

Constructors

- Constructors are used to initialize objects (set the attributes) as they are created.
- Different versions of the constructor can be implemented with different initializations e.g., one version sets all attributes to default values while another version sets some attributes to the value of parameters.
- **New term:** method overloading, same method name, different parameter list.

```
public Person(int anAge) {
    age = anAge;
    name = "No-name";
}
```

```
public Person() {
    age = 0;
    name = "No-name";
}
```

```
// Calling the versions (distinguished by parameter list)
Person p1 = new Person(100);    Person p2 = new Person();
```

James Tam

Example: Multiple Constructors

- **Name of the folder containing the complete example**:
  `fourth_constructorOverloading`

James Tam

Class Person

```
public class Person
{
    private int age;
    private String name;

    public Person()
    {
        System.out.println("Person()");
        age = 0;
        name = "No-name";
    }
```

James Tam

Class Person(2)

```
public Person(int anAge) {
    System.out.println("Person(int)");
    age = anAge;
    name = "No-name";
}

public Person(String aName) {
    System.out.println("Person(String)");
    age = 0;
    name = aName;
}

public Person(int anAge, String aName) {
    System.out.println("Person(int,String)");
    age = anAge;
    name = aName;
}
```

James Tam

Class Person (3)

```
public int getAge() {
    return(age);
}

public String getName() {
    return(name);
}

public void setAge(int anAge) {
    age = anAge;
}

public void setName(String aName) {
    name = aName;
}
}
```

James Tam

Class Driver

```
Person()
Person(int)
Person(String)
Person(int,String)
```

```
public class Driver {
    public static void main(String [] args) {
        Person jim1 = new Person();  // age, name default
        Person jim2 = new Person(21);   // age=21
        Person jim3 = new Person("jim3");  // name="jim3"
        Person jim4 = new Person(65,"jim4");
        // age=65, name = "jim4"

        System.out.println(jim1.getAge() + " " +
          jim1.getName());
        System.out.println(jim2.getAge() + " " +
          jim2.getName());
        System.out.println(jim3.getAge() + " " +
          jim3.getName());
        System.out.println(jim4.getAge() + " " +
          jim4.getName());
    }
}
```

```
0 No-name
21 No-name
0 jim3
65 jim4
```

James Tam

New Terminology: Method Signature

- Method signatures consist of: the type, number and order of the parameters.
- The signature will determine the overloaded method called:

```
Person p1 = new Person();
Person p2 = new Person(25);
```

James Tam

Overloading And Good Design

- Overloading: methods that implement similar but not identical tasks.
- Examples include class constructors but this is not the only type of overloaded methods:

```
System.out.println(int)
System.out.println(double)
```

etc.

For more details on class `System` see:

- http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintStream.html

- Benefit: just call the method with required parameters.

James Tam

Method Overloading: Things To Avoid

- Distinguishing methods solely by the order of the parameters.

```
m(int,char);
```

Vs.

```
m(char,int);
```

- Overloading methods but having an identical implementation.
- Why are these things bad?

James Tam

Method Signatures And Program Design

- Unless there is a compelling reason do not change the signature of your methods!

Before:

```
class Foo
{
    void fun()
    {

    }
}
```

After:

```
class Foo
{
    void fun(int num)
    {

    }
}
```

```
public static void main ()
{
    Foo f = new Foo();
    f.fun()
}
```

This change has broken me! ⊗

James Tam

New Terms And Definitions

- Method vs. Function
- Accessor method (“get”)
- Mutator method (“set”)
- Method overloading
- Method signature

James Tam

After This Section You Should Now Know

- What are accessor and mutator methods and how they can be used in conjunction with encapsulation
- What is method overloading and why is this regarded as good style

James Tam

Copyright Notification

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."

James Tam