datamining
apps

Learn more about our research, discover data science, and find other great resources at:

http://www.dataminingapps.com

Chapter 7
Delving Further into Object-Oriented Concepts

Overview

- Annotations
- Overloading methods
- The `this` keyword
- Information Hiding
- Class Inheritance
- Packages
- Interfaces
- Garbage Collection

Annotations

- Provide metadata about source code
- Inspected and evaluated by the compiler
- Begin with @
- Examples:
  - `@Test`: cf. JUnit Testing
  - `@Deprecated`: indicates that an element should no longer be used
  - `@Override`: subclass is overriding an element from a superclass
  - `@SuppressWarnings`: instructs the compiler to suppress specific warnings

Overloading Methods

- Using the same name for more than one method in the same class

```
public class Book {
  String title;
  String author;
  boolean isRead;
  int numberOfReadings;

  public void read(){
    isRead = true;
  }

  public void read(){
    numberOfReadings++;
}}
```

ERROR!

Overloading Methods

```
public class Book {
  String title;
  String author;
  boolean isRead;
  int numberOfReadings;

  public void read(){
    isRead = true;
    numberOfReadings++;
}

  public void read(int i){
    isRead = true;
    numberOfReadings += i;
}}
```

Overloading methods

```
public class Book {
String title;
String author;
boolean isRead;
int numberOfReadings;

public Book(String bookTitle){
title = bookTitle;
author = "Unknown";
}

public Book(String bookTitle, String
authorName){
title = bookTitle;
author = authorName;
}

…
```

```
public Book(String bookTitle, String
authorName, boolean hasBeenRead){
title = bookTitle;
author = authorName;
isRead = hasBeenRead;
}
public Book(String bTitle, String aName,
boolean hasBeenRead, int read){
title = bTitle;
author = aName;
isRead = hasBeenRead;
numberOfReadings = read;
}
}
```

The `this` keyword

- Refer to the object whose method is being called

```
public class Person {
  String name;

  public Person(String name){
    this.name = name;
  }

  public void printName(){
    System.out.println(name);
}}
```

The this keyword

```
public class Person {
  String name;

  public Person(){
    this("Unknown");
  }
  public Person(String name){
    this.name = name;
  }

  public void printName(){
    System.out.println(name);
}
}
```

The `this` keyword

```
import java.time.*;

public class Person {
String name;
LocalDate birthdate;

public Person(String name, int year, int month, int
day) {
this.name = name;
this.birthdate = LocalDate.of(year, month, day);
}

public Person(String name) {
this(name, 1900, 1, 1);
}

public int calculateAge() {
Period p = Period.between(this.birthdate,
LocalDate.now());
return p.getYears();
}
…
```

```
public int calculateAge(LocalDate
date) {
Period p =
Period.between(this.birthdate, date);
return p.getYears();
}

public int calculateAge(int year, int
month, int day) {
Period p =
Period.between(this.birthdate,
LocalDate.of(year, month, day));
return p.getYears();
}
}
```

Information Hiding

- Aka encapsulation
- Hiden representation of objects by making member variables private
- Accessor methods
- Idea is to control the access to variables
- Improve program maintenance

Information Hiding

```
import java.math.BigDecimal;

public class Product {
String productName;
String productID;
BigDecimal productPrice;

public Product(String name, String id, String
price) {
this.productName = name;
this.productID = id;
this.productPrice = new BigDecimal(price);
}

public String displayString() {
return "Product " + this.productID + ": " +
this.productName
+ " costs " + this.productPrice;
}}
```

```
public class ProductProgram {

public static void main(String[] args) {
Product myWidget = new
Product("Widget","WID0001","11.50");
myWidget.productPrice = new BigDecimal("-
5.00");
System.out.println(myWidget.displayString
());

}
}
```

Information Hiding

- Access modifiers
  - `public`: can be accessed by any class
  - `protected`: can be accessed by subclasses or classes in the same package
  - no modifier: can be accessed by classes in the same package
  - `private`: can be accessed only from within the same class

	Class	Package	Subclass	World
Public	+	+	+	+
Protected	+	+	+	-
No modifier	+	+	-	-
Private	+	-	-	-

Information Hiding

- Getters are used to access variables's values for viewing only

```
import java.math.BigDecimal;

public class Product {
private String productName;
private String productID;
private BigDecimal productPrice;

public Product(String name, String id, String
price) {
this.productName = name;
this.productID = id;
this.productPrice = new BigDecimal(price);
}
…
```

```
public String getName(){
return this.productName;
}
public String getID(){
return this.productID;
}
public BigDecimal getPrice(){
return this.productPrice;
}
public String displayString() {
return "Product " + this.getID() + ": " +
this.getName()
+ " costs " + this.getPrice();
}
}
```

Information Hiding

- Setters are used to modify variable values

```
import java.math.BigDecimal;
public class Product {
private static BigDecimal minPrice = new BigDecimal("0.00");
private static BigDecimal maxPrice = new BigDecimal("999.99");
private String productName, productID;
private BigDecimal productPrice;

public Product(String name, String id, String price) {
this.setName(name);
this.setID(id);
this.setPrice(price);}

public String getName(){
return this.productName;}

public void setName(String name){
this.productName = name;}

public String getID(){
return this.productID;}

private void setID(String id){
this.productID = id;}
```

```
public BigDecimal getPrice(){
return this.productPrice;}

public void setPrice(String price) throws
IllegalArgumentException{
BigDecimal tempPrice = new BigDecimal(price);
if (!isValidPrice(tempPrice)){
throw new IllegalArgumentException(price);}
this.productPrice = tempPrice;}

public boolean isValidPrice(BigDecimal price){
if (price.compareTo(minPrice)<0){
return false;}
if (price.compareTo(maxPrice)>0){
return false;}
return true;}

public String displayString() {
return "Product " + this.getID() + ": " +
this.getName()
+ " costs " + this.getPrice();
}}
```

Information Hiding

```
public class ProductProgram {

public static void main(String[] args) {
Product myWidget = new Product("Widget","WID0001","11.50");
try {
myWidget.setPrice("-5.0");
} catch (IllegalArgumentException e){
System.out.println(e + " is an invalid price.");
}
System.out.println(myWidget.displayString());
}
}
```

Information Hiding

```
public class ProductProgram {

public static void main(String[] args) {
Product myWidget = new Product("Widget","WID0001","11.50");
try {
myWidget.setPrice("-5.0");
} catch (IllegalArgumentException e){
System.out.println(e + " is an invalid price.");
}
System.out.println(myWidget.displayString());
}
}
```

Output:

java.lang.IllegalArgumentException: -5.0 is an invalid price.
Product WID0001: Widget costs 11.50

Information Hiding

```
import java.math.BigDecimal;

public class Account {
private String name;
private BigDecimal amount;

public Account(String acctName, String startAmount)
{
setName(acctName);
setAmount(startAmount);
amount.setScale(2, BigDecimal.ROUND_HALF_UP);}

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

public BigDecimal getAmount() {
return this.amount;}

public void setName(String newName) {
String pattern = "^[a-zA-Z0-9]*$";
if (newName.matches(pattern)) {
this.name = newName;}
}

private void setAmount(String newAmount){
this.amount = new BigDecimal(newAmount);}
```

```
public void withdraw(String withdrawal) throws
IllegalArgumentException{
BigDecimal desiredAmount = new BigDecimal(withdrawal);
//if desired amount is negative, throw an exception
if (desiredAmount.compareTo(BigDecimal.ZERO) < 0){
throw new IllegalArgumentException();
}
//if the amount is less than the desired amount, throw
an exception
if (amount.compareTo(desiredAmount) < 0){
throw new IllegalArgumentException();
}
this.amount = this.amount.subtract(desiredAmount);
}

public void deposit(String deposit) throws
IllegalArgumentException{
BigDecimal desiredAmount = new BigDecimal(deposit);
//if desired amount is negative, throw an exception
if (desiredAmount.compareTo(BigDecimal.ZERO) < 0){
throw new IllegalArgumentException();
}
this.amount = this.amount.add(desiredAmount);}
}
```

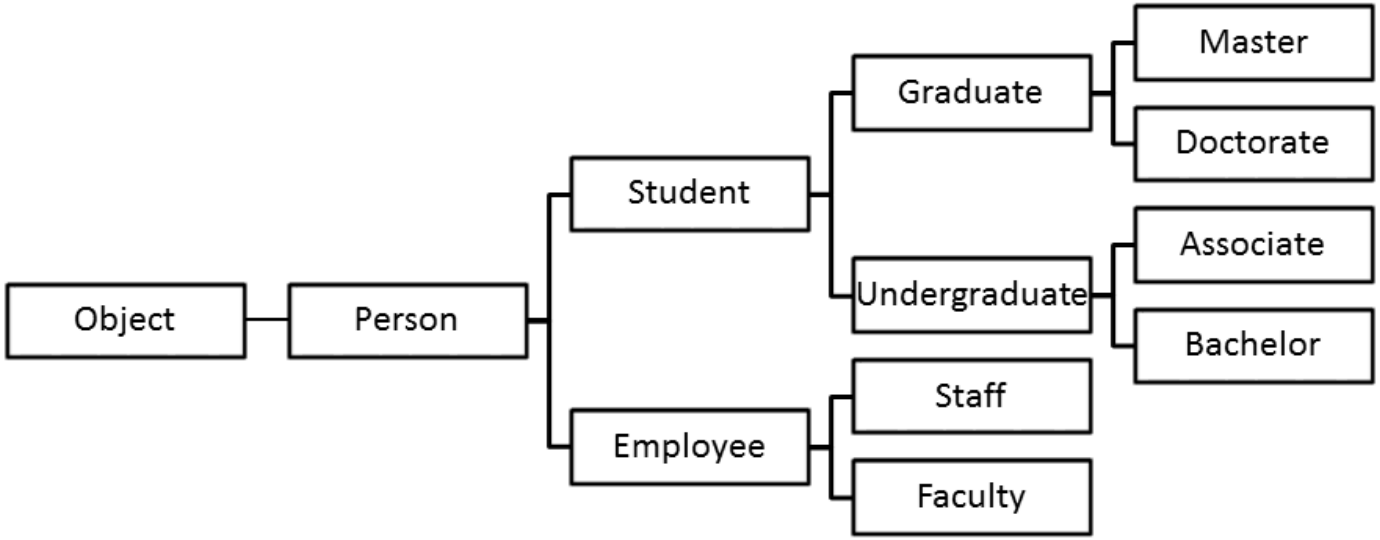
Information Hiding

```
public class AccountManager {

public static void main(String[] args) {
Account myAccount = new Account("FirstAccount","10.00");
System.out.println("Account Created: " + myAccount.getName());
System.out.println("Balance: " + myAccount.getAmount());
try {
myAccount.withdraw("20.00");
} catch (IllegalArgumentException e) {
System.out.println("Invalid Withdrawal");
} finally {
System.out.println("New Balance: " + myAccount.getAmount());
}}}
```

Class Inheritance

- Java classes are organized in a hierarchical inheritance structure
- Subclasses are derived from Superclasses
- Class `Object` is the highest-level superclass
- Inheritance represents “is a” relationship
  - Superclass: Person
  - Subclass: Student
  - Subclass: Employee
- `public class Employee extends Person{ }`

Class Inheritance



Class Inheritance

- Super refers to the superclass

```
public class Person {
private String name;

public Person(String name){
this.setName(name);
}

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

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

```
public class Employee extends Person {
private Employee manager;
private int id;

public Employee(String name, Employee manager, int
empID) {
super(name);
this.setManager(manager);
this.setEmployeeID(empID);}

public Employee getManager() {
return manager;}

public void setManager(Employee manager) {
this.manager = manager;}

public int getEmployeeID() {
return id;}

private void setEmployeeID(int employeeID) {
this.id = employeeID;}
}
```

Class Inheritance

```
public String displayName(){
return "Employee: " + super.getName();
}


public String displayIdentification() {
return "Person: " + super.id + " has EmployeeID " +
this.id;
}
```

Class Inheritance

- Subclasses can override methods of superclass

Student Superclass

```
public double calculateGPA() {
double sum = 0;
int count = 0;
for (double grade : this.grades){
sum += grade;
count++;
}
return sum/count;
}
```

Graduate Subclass

```
@Override
public double calculateGPA(){
double sum = 0;
int count = 0;
for (double grade :
this.getGrades()){
if (grade > minGrade){
sum += grade;
count++;
}}
return sum/count;
}
```

Class Inheritance

- Static methods can be bound to their implementations at compile time (static binding)
- Static methods cannot be overridden (they can be overloaded!)
- Dynamic binding (aka virtual method invocation), is the binding used for instance methods to allow for polymorphism
- Dynamic binding means that the binding of instance methods to the appropriate implementation is resolved at runtime, not at compile time, based on the object and its type
- JVM performs bottom up search in class hierarchy for method implementation

Class Inheritance

```
public class PersonProgram {

public static void main(String[] args){

Student john = new Master("John Adams");
john.setGrades(0.75,0.82,0.91,0.69,0.79);
Student anne = new Associate("Anne Philips");
anne.setGrades(0.75,0.82,0.91,0.69,0.79);

System.out.println(john.getName() + ": " + john.calculateGPA());
System.out.println(anne.getName() + ": " + anne.calculateGPA());
}
}
```

Output:

John Adams: 0.865

Anne Philips: 0.792

Class Inheritance

- Superclass Object
  - `protected Object clone()`: Creates and returns a copy of this object
  - `public boolean equals(Object obj)`: Indicates whether some other object is “equal to” this one
  - `protected void finalize()`: Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
  - `public final Class getClass()`: Returns the runtime class of an object
  - `public int hashCode()`: Returns a hashcode value for the object
  - `public String toString()`: Returns a string representation of the object

Class Inheritance

- Abstract classes
  - Cannot be instantiated
- Abstract methods
  - Declared without an implementation

```
public abstract class Shape {
private String color;

public Shape(String color) {
this.setColor(color);}

public String getColor() {
return this.color;}

public void setColor(String color) {
this.color = color;}

public abstract double calculateArea();
}
```

Class Inheritance

```
public class Circle extends Shape {
private double radius;
private final double PI = 3.14159;

public Circle(String color, double radius) {
super(color);
this.setRadius(radius);}

public double getRadius() {
return this.radius;}

private void setRadius(double radius) {
this.radius = radius;}

@Override
public double calculateArea() {
return PI * this.getRadius() * this.getRadius();
}}
```

Packages

- Grouping of related classes with features such as namespace and access control
- `import` statement
  - E.g. `import java.math.BigDecimal;`
- Include in CLASSPATH environment variable

Interfaces

- An interface defines a protocol, or contract, of behavior
- Template for a class, as it specifies what a class must do, but not how to do it
- `public interface InterfaceName`
- Interfaces contain headings for public methods without implementations
- An interface may have variables only if they are final, static, and initialized with a constant value.

Interfaces

- a class can implement an interface
  - must implement all methods defined in the interface
- a single class can implement any number of interfaces
- Interfaces can extend one or more other interfaces
- Polymorphism applies to interfaces similar to superclasses.

Interfaces

```
/**
* An interface for measuring methods.
*/
public interface Measurable{
/** Returns the perimeter of an object */
double calculatePerimeter();
/** Returns the area of an object */
double calculateArea();
}
```

Interfaces

```
public abstract class Shape implements Measurable {
private String color;

public Shape(String color) {
this.setColor(color);}

public String getColor() {
return this.color;}

public void setColor(String color) {
this.color = color;}
}
```

Interfaces

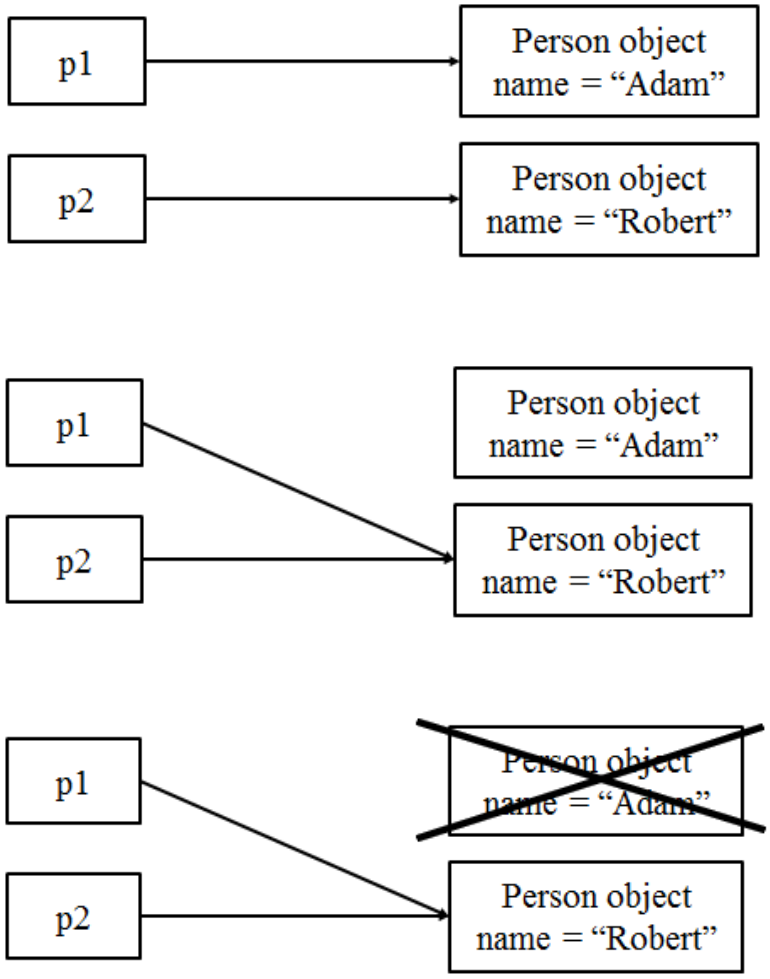
```
public class Circle extends Shape {
private double radius;
private final double pi = 3.14159;

public Circle(String color, double
radius) {
super(color);
this.setRadius(radius);
}

public double getRadius() {
return this.radius;
}

private void setRadius(double
radius) {
this.radius = radius;
}

@Override
public double calculateArea() {
return pi * this.getRadius() *
this.getRadius();
}

@Override
public double calculatePerimeter() {
return 2 * pi * this.getRadius();
}
}
```

Garbage Collection

- Garbage collection is a way to reclaim memory from objects once they are no longer in use
- Java has automatic garbage collection
- `finalize` method
  - invoked just before the object is destroyed

Garbage Collection

```
class PersonManager {
public static void main(String args[]) {
Person p1 = new Person("Adam");
Person p2 = new Person("Robert");
p1 = p2;
...//Rest of program
}
}
```



Conclusions

- Annotations
- Overloading methods
- This keyword
- Information Hiding
- Class Inheritance
- Packages
- Interfaces
- Garbage Collection