Object Orientation Part 1 Overview, Java

|  |  |
| --- | --- |
| **Objected Orientation**  **Class** defines the data and methods associated with the class  May support:  **Class Variables**  data that belongs to the class as a whole instead of instances of the class. There is only one occurrence of class variables.  **Instance Variables**  data that belongs to individual objects. Each instance object has its own copy of these instance variables.  **Class Methods**  functions that are associated with the class as a whole instead of instances of the class. These methods can reference class variables, but cannot reference instance variables since class methods are not associated with an instance.  **Instance Methods**  functions that are associated with instances of the class. These methods can access instance variables associated with an individual instance. These can also reference class variables.  **Constructor** a function responsible for creating an instance of a Class. It also is responsible for initializing that instance.  **Object** an individual instance of a class.  **Iterator** provides easy access to each item from an object that supports multiple items | When analyzing a business problem, classes and instance variables are nouns. Instance variables are often the characteristics describing a class. Methods are verbs.  Suppose we are representing the important OO classes at a university:  Student - student data and corresponding methods. Instance variables:  Abc123id, firstName, lastName, classification, gpa, totalGradePoints, totalGradeHours, major, minor, //address?  Course - information about a course in the catalog. Instance variables:  courseNr, description, (\*) prereq, creditHours, title,  Offering - an offering of a course for a particular semester. Instance variables:  courseNr, section, semester, days, startTime, endTime, prof  Enrollment - an association between a student and an Offering. Instance variables:  Abc123, courseNr, section, semester, grade  What class method(s) might we include with the Course class? (not with an instance of Course)  listCourses,  What instance method for Student?  Apply vs createStudent, admit, evaluateDegree, listEnrollments, withdrawFromUniversity, validatePassword, modifyAddress,  What instance methods for Enrollment?  Add, withdraw, recordGrade, |
| **OO in Java**   * Uses a file named *className* .**java** to create the class *className* as a public class. * A .java file can contain multiple class definitions, but only one should be a public class. * Objects are created as **dynamics** (heap memory) * **Garbage collection** is used to recover no longer referenceable objects. * Objects can inherit methods and variables from a **superclass** * **Interfaces** which specify method signatures and variables are provided to formally provide polymorphism * Java provides controls on inheritance via   **private** not known outside the class  **public** known inside class, in subclasses, and outside  **protected** known inside, known in subclasses, not known outside  **Default** known inside package   * **Generics**  are used to provide generic typing of classes. * **Reflection** is provided to examine metadata at runtime and dynamically use methods and variables at runtime. | *controlModifier* class *ClassName*  {  // attributes  // constructor methods  // class methods  // instance methods  } |
| **Defining a Class - Java**  The constructor methods have the same name as the class. You can have multiple constructor methods when different parameters are needed.  The **static** keyword is used to define class variables and class methods. //class variable-> one occurrence of | public class Employee  {  private static long lastId = 0; // last generated id  private String name;  private String namePrefix;  public double salary;  private int exemptions;  private long id;  // Constructor  public Employee(String prefix, String nm, double pay)  {  namePrefix = prefix;  name = nm;  salary = pay;  exemptions = 1;  id = generateId();  }  public Employee(String nm, double pay)  {  this.namePrefix = "";  this.name = nm;  this.salary = pay;  this.exemptions = 1;  this.id = generateId();  }  private static long generateId()  {  if (lastId == 0)  lastId = 111;  lastId++;  return lastId;  }  public String getFormattedName()  {  if (namePrefix.equals(""))  return name;  else  return namePrefix + " " + name;  }  public double calculatePay()  {  double pay = salary / 26;  if (exemptions == 1)  return pay;  else  return pay \* 0.9;  }  } |
| **Program Structure in Java**  Java programs consist of classes that are grouped into packages. Multiple classes can be included into one file with the following restrictions:   * Within a file, at most one class can be public. * The filename must be named after that public class. * Execution starts at a method with this signature:   public static void main (String[] args)   * Multiple files can be included in the same **package**.     Packages can help prevent name collisions. For the example, both packages could have a class named Location. |  |
|  |  |
| **Important Concepts - Inheritance**  **Inheritance**  a class can be defined as a subclass of another class allowing it to inherit variables and methods from the superclass. The subclass has a "is-a" relationship to the superclass.  In the example, Manager is an Employee. It inherits the instance variables: name, namePrefix, exemptions, salary, and id. Manager has its own additional attribute bonus. | public class Manager extends Employee //has all attributes from employee  {  private double bonus;  public Manager (String prefix, String nm, double pay, double mgrBonus)  {  super(prefix, nm, pay);  bonus = mgrBonus;  }  public double calculatePay()//overrides method of same name from employee  {  double basePay = super.calculatePay();  return basePay + bonus/12;  }  }  // To test those classes:  public class MainClass  {  public static void main(String [] args)  {  Employee emp = new Employee("Mr.", "Bob Wire",36000.00);  System.out.printf("Name = %s, Pay=%7.2f"  ,emp.getFormattedName(), emp.calculatePay());  Manager mgr = new Manager("Dr.", "R. Pepper", 50000.00, 500.00);  System.out.printf("Name = %s, Pay=%7.2f"  ,mgr.getFormattedName(), mgr.calculatePay());  System.exit(0);  }  } |
| **Important Concepts - Polymorphism**  **Polymorphism**  With polymorphism, subclasses can have their own implementation of a commonly named method. This allows a caller to use the common method name. Subclasses can override the implementation of a superclass' method. | Employee and Manager use the same getFormattedName method. Additionally, both have a calculatePay, but Manager overrode it. Notice that emp is first an Employee and then it is a manager.  // To test those classes:  public class MainClass  {  public static void main(String [] args)  {  Employee emp;  emp = new Employee("Mr.", "Bob Wire",36000.00);  System.out.printf("Name = %s, Pay=%7.2f\n"  ,emp.getFormattedName(), emp.calculatePay());  emp = new Manager("Dr.", "R. Pepper", 50000.00, 500.00);  System.out.printf("Name = %s, Pay=%7.2f\n"  ,emp.getFormattedName(), emp.calculatePay());  System.exit(0);  }  } |
| **Important Concepts - Encapsulation**  **Encapsulation** (aka, Information Hiding)  the internals of a class design can be hidden from the public perspective | // Suppose we have a StackOne class which uses an array to  // implement the stack  public class StackOne  {  private int [] dataM;  private int count = 0;  // Constructor  public StackOne (int capacity)  {  dataM = new int [capacity];  }  public void push(int item) throws StackOverflow  {  if (count == dataM.length)  {  throw new StackOverflow(count);  }  dataM[count] = item;  count++;  }  public int pop() throws StackUnderflow  {  if (count == 0)  throw new StackUnderflow();  count--;  return dataM[count];  }  public Boolean isEmpty()  {  return count==0;  }  public int size()  {  return count;  }  } |
| In this implementation of StackTwo, a linked list is used for the Stack. The actual implementation is hidden from consumers of StackOne and StackTwo.  What are some of the few public differences?  You ‘re not going to overflow.  No capacity passed in to constructor | // Suppose we have a StackTwo class which uses a linked list to  // implement the stack  public class StackTwo  {  private class Node // Node is only known to StackTwo  {  public int data;  public Node next;  public Node (int data, Node next)  {  this.data = data;  this.next = next;  }  }  private Node top = null;  private int count = 0;    // Constructor  public StackTwo ()  {  top = null;  count = 0;  }  public void push(int item)  {  top = new Node(item, top);  count++;  }  public int pop() throws StackUnderflow  {  if (top == null)  throw new StackUnderflow();  int item = top.data;  top = top.next;  count--;  return item;  }  public Boolean isEmpty()  {  return top==null;  }  public int size()  {  return count;  }  } |
| **Exception Classes**  In Java, we can create exception classes which can maintain information relevant to that exception. | public class StackOverflow extends Exception  {  public int capacity;  // constructor  public StackOverflow(int count)  {  capacity = count;  }  public StackOverflow()  {  capacity = -1;  }  public String toString()  {  if (capacity == -1)  return "Stack Overflow".toString();  else  {  StringBuffer sb = new StringBuffer();  sb.append("Stack Overflow, capacity is ");  sb.append(Integer.toString(capacity));  return sb.toString();  }  }  } |
| **Important Concepts - Generic**  **Generic**  Ability to specify the data type associated with a feature of a Class. For example, a Stack class may allow specification of the type of data stacked.  We could have originally described the kind of thing the Stack is storing as an **Object**, but what does the **generic** provide that isn't provided by specifying Object?  You’re being very specific with the data type. If another data type -> error. <> determines the data type. If you want object -> <object>.  Consumer:  StackThree<String> stack3 = new StackThree<String>();  stack3.push("Maynard");  stack3.push("Robbins"); | // Suppose we have a StackThree class which uses a linked list to  // implement the stack of a specified type. StackThree allows  // a generic specification of the type for its data  public class StackThree<T>  {  private class Node // Node is only known to StackThree  {  public T data;  public Node next;  public Node (T data, Node next)  {  this.data = data;  this.next = next;  }  }  private Node top = null;  private int count = 0;    // Constructor  public StackThree ()  {  top = null;  count = 0;  }  public void push(T item)  {  top = new Node(item, top);  count++;  }  public T pop() throws StackUnderflow  {  if (top == null)  throw new StackUnderflow();  T item = top.data;  top = top.next;  count--;  return item;  }  public Boolean isEmpty()  {  return top==null;  }  public int size()  {  return count;  }  } |
| **Important Concepts - Interface**  An interface describes the public information (methods, variables) provided.  This allows polymorphism for methods defined by that interface and the definition of a class to inherit from another class. | public interface StackInterface<T>  {  void push(T item);    T pop() throws StackUnderflow;  Boolean isEmpty();  int size();  } |
| **Defining a Class Which Implements an Interface** | public class StackTwo implements StackInterface<Integer>  {  private class Node  {  public Integer data;  public Node next;  public Node (Integer data, Node next)  {  this.data = data;  this.next = next;  }  }  private Node top = null;  private int count = 0;    // Constructor  public StackTwo ()  {  top = null;  count = 0;  }  public void push(Integer item)  {  top = new Node(item, top);  count++;  }  public Integer pop() throws StackUnderflow  {  if (top == null)  throw new StackUnderflow();  Integer item = top.data;  top = top.next;  count--;  return item;  }  public Boolean isEmpty()  {  return top==null;  }  public int size()  {  return count;  }  } |
| **Example of when Interfaces are Useful**  The Array.sort(*items*) method requires that *items* implements the Comparable<T> interface.  Suppose we want to sort instances of the Employee class on salary.  import java.util.\*;  ...  {  Employee[] employeeM = new Employee[3];  employeeM[0] = new Employee("Col", "Pop Corn", 75000);  employeeM[1] = new Employee("Dr", "Rob Benz", 125000);  employeeM[2] = new Employee("Ms", "Mae Nard", 80000);  Arrays.sort(employeeM);  for (Employee emp : employeeM)  System.out.printf("%9.2f %s\n"  , emp.salary  , emp.getFormattedName());  }  Output:  75000.00 Pop Corn  80000.00 Mae Nard  125000.00 Rob Benz | public class Employee implements Comparable<Employee>  {  private static long lastId = 0; // last generated id  private String name;  private String namePrefix;  private double salary;  private int exemptions;  private long id;  // Constructor  public Employee(String prefix, String nm, double pay)  {  namePrefix = prefix;  name = nm;  salary = pay;  exemptions = 1;  id = generateId();  }  private static long generateId()  {  if (lastId == 0)  lastId = db.getLastId();  lastId++;  return lastId;  }  public String getFormattedName()  {  if (namePrefix != "")  return name;  else  return namePrefix + " " + name;  }  public double calculatePay()  {  double pay = salary / 26;  if (exemptions == 1)  return pay;  else  return salary \* 0.9;  }  public int compareTo(Employee other)  { // compare the double values returning <0, 0 or >0  return Double.compare(salary, other.salary);  }  } |
| **Reflection in Java**  **Reflection** is provided to examine metadata at runtime and dynamically use methods and variables at runtime.  Why use reflection?   * Check whether an instance belongs to a class. * Create utility software which needs to access the details about particular instances * Dynamically create an instance of a class that is not known at compile time. The class name might come from a configuration file.   Some useful capabilities:  *instance* **instanceof** *className*  **instanceof** returns True if the instance is of that class or a subclass of that class | // Checks the class of an instance  import java.lang.reflect.\*;  public class InheritMainClass  {  public static void main(String [] args)  {  Employee emp;  Class metaClass;  emp = new Employee("Mr.", "Bob Wire",36000.00);  if (emp instanceof Employee)  System.out.printf("Employee\n");  if (emp instanceof Manager)  System.out.printf("Manager\n");  System.out.printf("Name = %s, Pay=%7.2f\n"  ,emp.getFormattedName(), emp.calculatePay());  emp = new Manager("Dr.", "R. Pepper", 50000.00, 500.00);  if (emp instanceof Employee)  System.out.printf("Employee\n");  if (emp instanceof Manager)  System.out.printf("Manager\n");  System.out.printf("Name = %s, Pay=%7.2f\n"  ,emp.getFormattedName(), emp.calculatePay());  System.exit(0);  }  }  **Output:**  **Employee**  **Name = Mr. Bob Wire, Pay=1384.62**  **Employee**  **Manager**  **Name = Dr. R. Pepper, Pay=2423.08** |
| **Utility Software to print the contents of objects**  Suppose you have to write code that will need to reference classes that your software has no knowledge of in advance. The utility needs to print each of the attributes for the object.  Without reflection, we would have each object print its own information. If feasible, that would be preferred; however, it may be less expensive to have a utility do the work.  Some useful capabilities (continued):  Class *metaClass = instance.***getClass()**;  **getClass()** returns a **Class** object which provides metadata information  *metaClass.***getName**()  **getName()** returns the name of the class instance  *metaClass.***getDeclaredFields**()  **getDeclaredFields()** returns an array of the attributes (aka, fields) of the clas. The entries are of class **Field**. This doesn't include attributes from the superclass.  *metaClass.***getSuperclass**()  **getSuperclass()** returns a **Class** object which provides metadata information about the superclass  *field*.**getType()**  **getType()** returns the datatype of a field.  *field*.**get(***instance***)**  **get(***instance***)** returns the value of a field when passed a particular instance of the object which contains that field. | // A utility to output the attributes of an object. It doesn't attempt  // to handle associated objects except Strings.  import java.lang.reflect.\*;  public class PrtObjUtil  {  public static void printObj(Object obj)  {  Class metaClass = obj.getClass();  System.out.printf(">>>" + metaClass.getName() + "\n");  // Since some attributes may be in the superclass,  // loop until we are done with all the classes  while (metaClass != null)  {  // get the attributes  Field [] fieldM = metaClass.getDeclaredFields();  // Make even private attributes available  AccessibleObject.setAccessible(fieldM, true);  // loop through each of the attributes  for (Field f : fieldM)  {  try  {  // get the datatype and value  Class typ = f.getType();  Object val = f.get(obj);  // Print primitve and String attributes  if (typ.isPrimitive()  || typ.isAssignableFrom(String.class))  {  String outString = f.getName() + "=" + val.toString();  System.out.printf("%s\n", outString);  }  }  catch (Exception e)  {  e.printStackTrace();  }  } // for  // get the superclass  metaClass = metaClass.getSuperclass();  } // while  } //printObj  } |
| Sample output for the program to the right  **>>>Employee**  **lastId=112**  **name=Bob Wire**  **namePrefix=Mr.**  **salary=36000.0**  **exemptions=1**  **id=112**  **>>>Manager**  **bonus=500.0**  **lastId=113**  **name=R. Pepper**  **namePrefix=Dr.**  **salary=50000.0**  **exemptions=1**  **id=113** | // Use that PrtObjUtil to print the attributes and their values  public class ReflectMainClass  {  public static void main(String [] args)  {  Employee emp;  emp = new Employee("Mr.", "Bob Wire",36000.00);  PrtObjUtil.printObj(emp);  emp = new Manager("Dr.", "R. Pepper", 50000.00, 500.00);  PrtObjUtil.printObj(emp);  System.exit(0);  }  } |
|  |  |