Murdoch
UNIVERSITY

Topic 4
Programming with Methods

ICT167 Principles of Computer Science

Murdoch UNIVERSITY

Objectives

- Understand the use of **private** helper methods hidden from the **public** interface of a class
- Know that the calling object can be omitted if it is the same as the calling object in the invoking method
- Know that a **main** program can appear in any class including the class which it uses
- Understand how to use **static** variables and **static** methods

Objectives

- Explain the uses of **static** variables and **static** methods
- Understand the meaning and use of the reserved word **this** in Java
- Be able to use the common **static** methods in the **Math** class
- Explain what a **wrapper** class is and why it is used
- Be able to wrap and unwrap primitive values

Objectives

- Understand the **automatic boxing** and **unboxing** of wrapper classes in Java
- Make use of common methods in the wrapper classes
- Be able to design a Java program in a top-down manner
- Be able to test a complex program using driver programs and stubs

Reading – Savitch: Chapters 5.2, 6.2 – 6.3

Example Re-visited

- The class SpeciesFourthTry discussed in Topic 3 has the following attributes and methods:


```
// Class from Topic 3 - skeleton only
// includes equals method
import java.util.Scanner;
public class SpeciesFourthTry {
  // Instance variables
  private String name;
  private int population;
  private double growthRate;
```

Example Re-visited

```
// Methods
public void readInput()
{
   // …. code for the method readInput
}
public void writeOutput()
{
   // …. code for the method writeOutput
}
public int predictPopulation(int years)
{
   // …. code for the method
}
```

Example Re-visited

```
// Accessor or Get methods
public String getName()
{
   return name;
}
public int getPopulation()
{
   return population;
}
public double getGrowthRate()
{
   return growthRate;
}
```

Example Re-visited

```
// Mutator or Set method
public void setSpecies(String newName,
      int newPopulation, double
      newGrowthRate)
{
   // … code for the method set
}
```

Example Re-visited

```
   // Equals method: tests equality of 2 species
   public boolean equals(SpeciesFourthTry
                                          otherObject)
   {
      return
    ((this.name.equalsIgnoreCase(otherObject.name))
    &&(this.population == otherObject.population)
    &&(this.growthRate == otherObject.growthRate));
   }
} // end class SpeciesFourthTry
```

Example UML Class Diagram

SpeciesFourthTry
- name: String - population: int - growthRate: double
+ readInput(): void + writeOutput(): void + predictPopulation(): int + setSpecies(String newName, int newPopulation, double newGrowthRate): void + getName(): String + getPopulation(): int + getGrowthRate(): double + equals (SpeciesFourthTry otherObject): boolean

Example Client

```
import java.util.Scanner;
/** Client Program / Test program */
public class SpeciesFourthTryDemo {
   public static void main(String[] args) {
     SpeciesFourthTry s1= new SpeciesFourthTry();
     SpeciesFourthTry s2 =new SpeciesFourthTry();
     int numberOfYears, futurePopulation;
     System.out.println("Enter number of years:");
     Scanner keyboard = new Scanner(System.in);
     numberOfYears = keyboard.nextInt( );
     s1.readInput();
     s1.writeOutput();
```

Example Client

```
    futurePopulation = s1.predictPopulation(
                                        numberOfYears);
    // ……
    s2.setSpecies("Klingon ox", 10, 15);
    s2.writeOutput();
    if (s1.equals(s2))
      System.out.println("Two species are same");
    else
      System.out.println("Two species not same");
  }// end main
}// end SpeciesFourthTryDemo
```

Methods Calling Methods

- A method in a class might involve a lengthy or complex calculation
- So break it down into smaller parts
  - Use ***helper*** methods to perform some of the parts
- Helper methods can be declared to be **private**
  - They are not part of the public interface of the class: they are part of the implementation
- Look at the following example (class Oracle) from Savitch to see how helper methods are invoked

Example: Oracle Class

```
import java.util.Scanner;
public class Oracle {
   private String oldAnswer = "The answer is in your
    heart.";
   private String newAnswer;
   private String question;
   public static void main(String[] args) {
      Oracle delphi = new Oracle();
      delphi.chat();
   }// end main
   private void update() {
      oldAnswer = newAnswer;
   }
```

Example: Oracle Class

```
public void chat() {
    System.out.println("I am the oracle.");
    System.out.println("I will answer questions.");
    Scanner keyboard = new Scanner(System.in);
    String response;
    do {
        answer();
        System.out.println("Do you wish to ask
                                another question?");
        response = keyboard.next();
    }while(response.equalsIgnoreCase("yes"));
    System.out.println("Oracle will now rest.");
}
```

Example: Oracle Class

```
private void answer() {
    System.out.println("What is your question?");
    Scanner keyboard = new Scanner(System.in);
    question = keyboard.nextLine();
    seekAdvice();
    System.out.println("You asked the question:");
    System.out.println(" " + question);
    System.out.println("Now, here is my answer:");
    System.out.println(oldAnswer);
    update();
}
```

Example: Oracle Class

```
   private void seekAdvice() {
      System.out.println("I need some help on that.");
      System.out.println("Please give 1 line advice.");
      Scanner keyboard = new Scanner(System.in);
      newAnswer = keyboard.nextLine();
      System.out.println("Thanks. That helped lots.");
   }
} // end class Oracle
```

Methods Calling Methods

- See how the calling object can be omitted (when calling the helper method) if it is the same as the calling object in the invoking method
- The calling object `delphi` invokes its own method `chat()` as

```
delphi.chat();
```

- Within `delphi.chat()`, the helper method is invoked **without** using the name of the object and the dot notation, as follows:

Methods Calling Methods

```
public void chat () {
    ……
    answer();
    ……
}
```


- `answer()` is a helper (private) method which calls its two helper methods as follows:


```
private void answer() {
    ……
    seekAdvice();
    ……
    update();
}
```

Methods Calling Methods

- Also notice (in the `Oracle` class example) that the `main()` method has been placed inside the class which it uses
  - The `main()` method in this case acts like a client and is useful for testing purposes
- Note that within the `main()` method, you must create an object of the class before you can invoke any of the methods

Methods Calling Methods

- Eg: to invoke the method `chat()` within `main()`, you need to create an object of type Oracle in the usual way:


```
Oracle delphi = new Oracle();
```

- And then invoke a method of this object as:

```
delphi.chat();
```

- Since `main()` is a static method, it belongs to the class, and there will be only one `main()` method

Methods Calling Methods

- When user executes the program, the JVM (*Java Virtual Machine*) looks for the `main()` method.
- The `main()` method starts running, creates an object `delphi`, invokes its `chat()` method (which uses a helper method which in turn uses two helper methods), and gets the stuff done

The `reserved` Word `this`

- The reserved word `this` in Java stands for the name of the current (calling) object
  - That is, it refers to the object that contains the reference
- Methods called in an object definition file do not need to reference itself (the object)
- You may either use "`this.`", or omit it, since it is presumed
- For example, if `answer()` is a method defined in the class Oracle:

The **reserved** Word **this**

```
public class Oracle {
   ...
   public void chat() {
      ……
      // One way to invoke the answer() method
      // defined in this file is:
      // this.answer ();
      // Another way is to omit "this."
      answer ();   // "this." is presumed here
      ……
   }
   ...
} // end class Oracle
```

When an Object is Required

- Methods called *outside* the object definition require an object name to precede the method name
- For example:


```
Oracle delphi = new Oracle();
// delphi is not part of the
definition // code for Oracle
...
// chat is a method defined in Oracle
delphi.chat();
...
```

When an Object is Required

- Similarly in another program, the call to method `chat()` may be

```
Oracle myObject = new Oracle();
myObject.chat();
```

- And the call to method `answer ()` in this case would mean

```
myObject.answer();
```

`static` Variables

- When a Java program is running, if something is ***static*** then there is only one copy of it, no matter how many objects are created
- Static variables are shared by all objects of a class
  - Variables declared `static final` are considered constants – their values cannot be changed. Eg:

    ```
    public static final int UPPER_LIMIT = 999;
    ```

  - Variables declared `static` (without `final`) can be changed

    ```
    private static int counter;
    ```

`static` Variables

- Only one instance of the static variable exists which can be accessed by all objects of the class
- Static variables can be public or private – should normally be private and should be accessed or changed only by accessor and mutator methods
- Static variables are also called **class variables**
- **Therefore, Java has three kinds of variables: local variables, instance variables, and static variables**

Local, instance, and static variables

- **local variables:** declared in a method

- **instance variables:** declared in a class definition outside any method – belong to an object

- **static variables:** class variables - every object shares the one and only one

`static` Methods

- Some methods may have no relation to any type of object
  - Eg: a method to compute the maximum of two numbers or a method to find the square root of a number
- In such cases a method can be declared to be static

`static` Methods

- Eg:


```
public class MyClass {
…
public static boolean isPositive(int n){
   return (n>0);
}
…
} // end MyClass
```

`static` Methods

- The static method must still belong to a class
- It does not need a calling object - the class name is normally used instead during its invocation. Eg:

```
if (MyClass.isPositive(x))
 System.out.println("Positive");
```

- static methods are also called ***class methods***

`static` Methods

- Note that it **is** possible to create an object of `MyClass` and use it to invoke the `isPositive()` method, but doing so can be confusing to people reading your code
- Note that **all other methods** (*non-static*) must be part of an object, so an object must exist before they can be invoked
- Since a static method does not need a calling object, it **cannot** refer to a (non-static) instance variable of the class

`static` Methods

- Likewise, a static method **cannot** call a non-static method of the class (unless it creates an object of the class to use as a calling object)
- Use static methods:
  - For methods which do not involve an object
  - Small private helper methods in a class
  - Generally useful methods to do with numbers or Strings or input/output
    - Eg: finding the maximum of two numbers, computing a square root, generating a random number

`static` Methods

- Static methods are commonly used to provide libraries of useful and related methods
- Examples:
  - The *main* method in any class
  - The Math class
    - Automatically provided with Java
    - Methods include pow, sqrt, max, min, and many more methods

Example Class

```
// File: CircleCalculator.java
/** Class with static methods to perform calculations
    on circles. */
public class CircleCalculator {
   // constant
   public static final double PI = 3.14159;
   public static double getArea(double radius) {
      return (PI*radius*radius);
   }
   public static double getCircumference(double radius)
   {
      return (PI*(radius + radius));
   }
} // end class CircleCalculator
```

Example Client

```
// File: CircleCalculatorDemo.java
import java.util.Scanner;
public class CircleCalculatorDemo {
   public static void main(String[] args) {
      double radius;
      System.out.println("Enter the radius of a "
                                 + "circle in inches:");
      Scanner kb = new Scanner(System.in);
      radius = kb.nextDouble();
      System.out.println("A circle of radius " +
                                    radius + " inches");
```

Example Client

```
        System.out.println("has an area of " +
                           CircleCalculator.getArea(radius) +
                                                  " square inches,");
        System.out.println(" and circumference of "
        +CircleCalculator.getCircumference(radius)+
                                                  " inches.");
    } // end main
} // end class CircleCalculatorDemo
```

`static` Methods in Main Class

- The predefined class `Math` is automatically provided as part of the Java language, and contains a number of the standard mathematical methods
- All these methods are `static` and are called by using the class name `Math` in place of a calling object

`static` Methods in Main Class

- Eg:

```
System.out.println("The maximum of 5 and
                    7 is = " +
Math.max(5, 7));
Powers: Math.pow(2.0, 3.0) returns 8.0
Absolute value: Math.abs(-4) returns 4
                Math.abs(5) returns 5
              Math.abs(-5.1)returns 5.1
Maximum:   Math.max(5, 6) returns 6
Minimum:   Math.min(5.9, 6.5) returns 5.9
```

`static` Methods in Main Class

- Eg:

```
Rounding: Math.round(6.8) returns 7
          Math.round(6.49) returns 6
Ceiling: Math.ceil(3.2) returns 4.0
```


  - returns a whole number of type double
  - need to cast if you want an int. Eg:

```
int j = (int)Math.ceil(3.2);
```

```
Floor: Math.floor(3.2) returns 3.0
```

  - this too returns a whole number of type double, and need to type cast if you want an int

`static` Methods in Main Class

- Eg:

```
Square root: Math.sqrt(4.0) returns 2.0
Random:      Math.random() returns a random
             number greater than or equal to
             0.0 and less than 1.0
```


- See the on-line documentation for many more
- Note the `Math` class also contains some `static` constants such as `Math.PI` which is a `double` with value approximately equal to π.

NOTE: `main` method

- **You can put a *main* method in any class**
  - See class Oracle above in these slides
  - Usually **main** is by itself in a class definition
  - Sometimes it makes sense to have a **main** method in a regular class definition
  - When the class is used to create objects, the **main** method is ignored
  - Adding a diagnostic **main** method to a class makes it easier to test the class's methods

NOTE: `main` method

- **You can put a *main* method in any class**
  - Because main must be static, you cannot invoke non-static methods of the class in **main** unless you create an object of the class
  - Normally you would not put a **main** method in a class that is used to create objects unless it is for test purposes

Wrapper Classes

- As we know, Java treats primitive types and class types differently
  - Eg: the variables (arguments) of primitive types are passed to other methods using *call-by-value* whereas object variables are passed using *call-by-reference*
  - Similarly, the assignment operator == behaves differently for primitive types and for class types
- Occasionally we need to be able to make things uniform, and treat a primitive type as an object

Wrapper Classes

- Java has one special class associated with each primitive type - called **wrapper classes** - they "wrap up" the primitive data types as objects
  - Eg: there is an `Integer` class corresponding to `int`
  - Other wrapper classes include `Double`, `Long`, `Character` and `Boolean` corresponding to the primitive types `double`, `long`, `char` and `boolean`, respectively
  - All primitive types have an equivalent class

Wrapper Classes

- Why?
  - Some data structures which contain many things are designed to contain Objects only
  - The Wrapper classes have various useful methods, including ones to convert back to primitive types

Wrapper Classes

Primitive type	Class type	Method to convert back to primitive type
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()
char	Character	charValue()

Wrapper Classes

- Converting a primitive to a wrapper object, for example:


```
Integer n = new Integer(78);
```


  - declares an instance `n` of the `Integer` wrapper class with the value 78

- The object `n` is just an Object version of the number 78
- The `int` 78 is *wrapped up* as an Object belonging to the Class `Integer`

Wrapper Classes

- Unwrapping, for example:

```
int i = n.intValue();
```


  - the method `intValue` in the Class `Integer` returns the `int` which is wrapped up inside the wrapper object

- Similarly:

```
Double D = new Double(4.5);
double d = D.doubleValue();
```

Automatic Boxing and Unboxing

- Wrapping (converting/type casting) a value of a primitive to an object of its corresponding wrapper class is called ***boxing***
- Starting with Java 5.0, boxing is done automatically. Eg:

```
Integer n = 78;
```

- is equivalent to writing:

```
Integer n = new Integer(78);
```

Automatic Boxing and Unboxing

- Similarly, an object of a wrapper class can be converted to a value of a corresponding primitive type automatically (called automatic ***unboxing***)


```
int i = n;
```

- is equivalent to:

```
int i = n.intValue();
```

Automatic Boxing and Unboxing

- Note that automatic boxing and unboxing also apply to parameters
  - A primitive argument can be provided for a corresponding formal parameter of the associated wrapper class
  - A wrapper class argument can be provided for a corresponding formal parameter of the associated primitive type

Useful Constants and `static` Methods in Wrapper Classes

- `Integer.MAX_VALUE` returns the largest value allowed in type `int`
- Also, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE`, etc.
- Static methods in the wrapper classes can be used to convert a string to the corresponding number of type `int`, `long`, `float`, or `double`

Useful Constants and `static` Methods in Wrapper Classes

- Eg:

```
String str = "499.95";
double d = Double.parseDouble(str);
```


- or use:


```
Double.parseDouble(str.trim());
```

- if the string has leading or trailing whitespaces

Useful Constants and `static` Methods in Wrapper Classes

- Similarly:

```
String numString = "727";
int i = Integer.parseInt(numString);
long l = Long.parseLong(numString);
float r = Float.parseFloat("499.95");
```


- Methods for converting strings to the corresponding numbers are also available.

  Eg: `Integer.toString(78)`, `Long.toString(78)`, `Float.toString(499.95)`, and `Double.toString(499.95)`

Character Class `static` Methods

- The Character class wraps a char. Use:

```
Character c = new Character('a');
```


- to wrap a char

- Checks if c1 and c2 wrap the same char


```
c1.equals(c2);
// returns 'A'
Character.toUpperCase('a');
```

Character Class `static` Methods

- Eg:

```
char firstChar = 'a';
char secondChar =
  Character.toUpperCase(firstChar);
Character.toLowerCase('A')// returns 'a'
Character.isUpperCase('A')// returns
true
Character.isLowerCase('A')//returns
false
// returns false
Character.isWhitespace('A')
```

Character Class `static` Methods

- Eg:

```
// returns true if response is a digit
// character in the range 0 to 9 and
// false otherwise
Character.isDigit(response)
Character.isLetter('A') // returns true
Character.isLetter('?')// returns false
// returns the String "a"
Character.toString('a')
```

Top-Down Design

- = stepwise refinement = divide and conquer = breaking the problem down into smaller steps
- In pseudo-code, write a list of sub-tasks that the method must do
- If you can easily write Java statements for a sub-task, you are finished with that sub-task
- If you cannot easily write Java statements for a sub-task, treat it as a new problem and break it up into a list of sub-tasks

Top-Down Design

- Eventually, all of the sub-tasks will be small enough to easily design and code
- Solutions to sub-tasks might be implemented as private helper methods
- Top-down design is also known as *divide-and-conquer* or *stepwise refinement*

Top-Down Design

- Here is an example problem:
  - The user is given a list of items of various nett prices
  - Some items are 0% rated for the GST, call these category Z
  - The other items are rated at 10% for the GST, call these category G

Top-Down Design

- The user should enter the category of each item and then the price in cents
- The program should display the nett price, tax, and total cost of each item, and display a running total of tax and total cost
- The user can enter category ‘Q’ to finish
- Display all amounts in dollars and cents

Top-Level Pseudo-code

```
total = 0
totalTax = 0
cat = 'A'              //anything but 'Q'
while (cat != 'Q') {
  cat = get category letter from user
  if (cat !='Q'){
     price = get cents from user
     tax = taxOn( cat, price)
     cost = price + tax
     total = total + cost
     totalTax= totalTax + tax
```

Top-Level Pseudo-code

```
        //all values in cents
        DisplayInDollars("net price", price)
        DisplayInDollars("item tax", tax)
        DisplayInDollars("item cost", cost)
        DisplayInDollars("total tax", totalTax)
        DisplayInDollars("total cost", total)
    }//end if
}//end while
say goodbye
```


- In order to complete the description of the program we then need to consider the procedures which are used here

Tips for Writing Methods

- Apply the principle of encapsulation and detail hiding by using the public and private modifiers judiciously
  - If the user will need the method, make it part of the interface by declaring it public
  - If the method is used only within the class definition (a *helper* method, then declare it private)

Tips for Writing Methods

- Create a main method with diagnostic (test) code within a class's definition
  - Run just the class to execute the test/diagnostic program
  - When the class is used by another program the class's main method is ignored

Program Testing: Test Methods Separately

- Carefully test each method individually so you are (quite) sure that each method works correctly
  - Test programs are sometimes called ***driver programs***
  - **A driver program** is usually a main program (*main* method) designed only to test that a method works
  - Keep it simple: test only one new method at a time
    - Driver program should have only one untested method

Program Testing: Test Methods Separately

- If method A calls method B, then we think of method A being above method B. There are two approaches to testing:
- ***Top down testing***
  - Also called testing using *stubs*: test method A first and use a *stub* for method B
  - A *stub* is a method that stands in for the final version and does little actual work. It usually does something as trivial as printing a message or returning a fixed value. The idea is to have it so simple that you are nearly certain it will work

Program Testing: Test Methods Separately

- ***Bottom up testing***
  - Test method B fully (eg, using a driver program) before testing method A
  - Bottom-up testing means being sure that method B works before testing method A
  - Eg: check the procedure for getting a category letter from the user before checking the overall program

Example

- Here is a program including the category procedure and a driver program

```
import java.util.*;
public class CatTest {
   public static void main(String[] args) {
      // driver method for test purposes only
      char cat = 'a';
      while (true) {
         cat = getCat();
         System.out.println("Your category was " +cat);
      } //end of while
   } //end of main
```

Example

```
private static char getCat() {
    char c;
    Scanner kb = new Scanner(System.in);
    do {
        System.out.print("Enter a category Z, G or Q");
        c =  kb.next().charAt(0);
        c = Character.toUpperCase(c);
        if ((c != 'Z')&&(c != 'G')&&(c != 'Q'))
            System.out.println("Error-invalid category");
    } while ((c !='Z')&&(c!='G')&&(c!='Q'));
    return c;
} //end of getCat
} //end of class
```

Testing via Stubs

- Sometimes you want to test a large method before testing all the smaller methods which it calls
- For example, just to make sure that the overall approach looks promising
- Use a **stub** = a simplified version of a method for testing purposes
- Then just include a stub for any small methods which you have not developed or checked yet

Testing via Stubs

- Eg: here is a stub for DisplayInDollars()

```
private static void DisplayInDollars
                              (String msg, int cents) {
  System.out.println("DisplayInDollars Stub");
  System.out.println("Message is: " + msg);
  System.out.println("Cents value is: " + cents);
}
```


- At some later stage you can tidy this up
- So here is a half completed version of the whole program...

Example

```
import java.util.*;
public class GST {
   public static void main(String[] args) {
    int total = 0, totalTax = 0;
    char cat = 'A';      //anything but 'Q'
    while (cat != 'Q') {
       cat = getCat();
       if (cat != 'Q') {
          int price = getPrice();
          int tax = taxOn( cat, price);
          int cost = price + tax;
          total = total + cost;
          totalTax= totalTax+ tax;
```

Example

```
        //all values in cents
        DisplayInDollars("nett price", price);
        DisplayInDollars("item tax", tax);
        DisplayInDollars("item cost", cost);
        DisplayInDollars("total tax", totalTax);
        DisplayInDollars("total cost", total);
    }//end if
 }//end while
 System.out.println("good bye");
}//end main
```

Example

```
private static char getCat() {
   char c = 'A';
   Scanner kb = new Scanner(System.in);
   do {
      System.out.println("Enter a category - Z, G or
 Q:");
      c = kb.next().charAt(0);
      c = Character.toUpperCase(c);
      if ((c != 'Z')&&(c != 'G')&&(c != 'Q'))
         System.out.println("*Error-invalid category");
   }while((c != 'Z')&&(c != 'G')&&(c != 'Q'));
   return c;
}//end of getCat
```

Example

```
private static int getPrice() {
   System.out.println("** getPrice Stub **");
   System.out.println("Enter price in cents");
   Scanner kb = new Scanner(System.in);
   int cents = kb.nextInt();
   return cents;
}//end of getPrice

private static int taxOn(char cat, int price){
   if (cat == 'G' ) return price/10;
   else return 0;
}//end of taxon
```

Example

```
  private static void DisplayInDollars(String
                                       msg, int cents)
{
      System.out.println("*DisplayInDollars Stub*");
      System.out.println("Message is: " + msg);
      System.out.println("Cents value: " + cents);
  }//end of DisplayInDollars
}//end of class GST
```

Example

- And here is a complete version of the method DisplayInDollars:


```
private static void DisplayInDollars(String msg,
                                     int cents)
{
   String text;
   int dollars = cents / 100;
   cents = cents % 100;
   text = msg + " = $" + dollars + ".";
   if (cents < 10) text = text + "0" + cents;
   else text = text + cents;
   System.out.println(text);
}//end of DisplayInDollars
```

GST UML Class Diagram

GST
- category: char - priceInCents: int - taxInCents: int - costInCents: int
+ readCategory(): void + readPrice(): void + calculateTax(): void + calculateCost(): void + getCategory(): char + getPrice(): int + getTax(): int + getCost(): int + DisplayInDollarsInputData(): void + DisplayInDollars (String, int): void

Complete GST Class

- Here is a complete working version

```
// ICT167 Topic 4 Case Study in Program Design
// Object Oriented Version GSTv2 Class
// P S Dhillon

import java.util.*;

public class GSTv2 {
   // instance variables
   private char category;
   private int priceInCents;
   private int taxInCents;
   private int costInCents;
```

Complete GST Class

```
// input methods readCategory() and readPrice()
public void readCategory(){
    char c = 'A';
    Scanner kb = new Scanner(System.in);
    do {
       System.out.println("Enter a category-Z,G or Q:");
       c =  kb.next().charAt(0);
       c = Character.toUpperCase(c);
       if ((c != 'Z')&&(c != 'G')&&(c != 'Q'))
          System.out.println("*Error-invalid category");
    }while((c != 'Z')&&(c != 'G')&&(c != 'Q'));
    category = c;
}//end of getCat
```

Complete GST Class

```
public void readPrice() {
    System.out.println("Enter price in cents");
    Scanner kb = new Scanner(System.in);
    priceInCents = kb.nextInt();
}//end of getPrice
```

Complete GST Class

```
// calculate tax and cost methods
public void calculateTax() {
   if (category == 'G' )
      taxInCents = priceInCents/10;
   else taxInCents = 0;
}//end calculateTax

public void calculateCost() {
   costInCents = priceInCents + taxInCents;
}// end calculateCost
```

Complete GST Class

```
// get methods
public char getCategory() {
    return category;
}
public int getPrice() {
    return priceInCents;
}
public int getTax() {
    return taxInCents;
}
public int getCost() {
    return costInCents;
}
```

Complete GST Class

```
// output methods
public void DisplayInDollarsInputData() {
    DisplayInDollars("nett price ", priceInCents);
    DisplayInDollars("item tax ", taxInCents);
    DisplayInDollars("item cost ", costInCents);
}
```

Complete GST Class

```
public void DisplayInDollars(String msg, int
 cents){
    String text;
    int dollars = cents / 100;
    cents = cents % 100;
    text = msg + "=$" + dollars + ".";
    if (cents < 10) text = text + "0" + cents;
    else text = text + cents;
    System.out.println(text);
 }//end of DisplayInDollars
}//end class GSTv2
```

GST Client

```
// File: GSTv2Demo

public class GSTv2Demo {
   public static void main(String[] args) {
      // create a new object, call it: calculator
      GSTv2 calculator = new GSTv2();
      int totalCost = 0;
      int totalTax = 0;
      calculator.readCategory();
```

GST Client

```
while (calculator.getCategory() != 'Q') {
    calculator.readPrice();
    calculator.calculateTax();
    calculator.calculateCost();

    totalCost = totalCost+calculator.getCost();
    totalTax = totalTax+calculator.getTax();
```

GST Client

```
            // all values are in cents
            calculator.DisplayInDollarsInputData();
            calculator.DisplayInDollars("total tax ",
                                         totalTax);
            calculator.DisplayInDollars("total cost ",
                                         totalCost);

            calculator.readCategory();
        }//end while
        System.out.println("Good bye");
    }//end main
}//end GSTv2Demo class
```

End of Topic 4