

Deliverables


Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. To avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

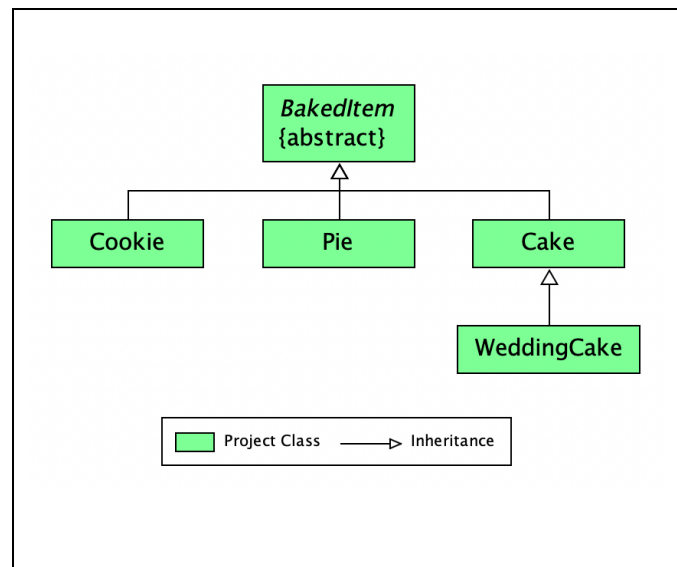
- BakedItem.java, *no test file (abstract class)* - methods should be tested in CookieTest.java
- Cookie.java, CookieTest.java
- Pie.java, PieTest.java
- Cake.java, CakeTest.java
- WeddingCake.java, WeddingCakeTest.java

Specifications

Overview: This project is the first of three that will involve a bakery and reporting for baked items. You will develop Java classes that represent categories of baked item including cookie, pie, cake, and wedding cake. Note that there is no requirement for a class with a main method in this project. You will need to create a JUnit test file for the indicated classes and write one or more test methods to ensure the classes and methods meet the specifications. You should create a

jGRASP project and add the source and test files as they are created. All your files should be in a single folder. The UML class diagram at right provides a visual overview of how the classes in the project relate to one another. As you add your classes to the jGRASP project, you should generate the UML class diagram by double-clicking  <UML> for the project in the Open Projects window. Once generated, you can use the mouse to select/drag a class to arrange the diagram similar to the one shown.

Alternatively, select BakedItem, right-click on it and select Layout > Tree Down, then select Dependents of Selected.



You should read through the remainder of this assignment before you start coding.

- **BakedItem.java**

Requirements: Create an *abstract* BakedItem class that stores BakedItem data and provides methods to access the data.

Design: The BakedItem class has fields, a constructor, and methods as outlined below.

- (1) **Fields:** *instance* variables for the BakedItem's name of type String, the flavor type String, the quantity of type int, and ingredients of type String[]; *static* (or class) variable of type int for the count of BakedItem objects that have been created (set to zero when declared and incremented in the constructor). These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of BakedItem. These are the only fields that this class should have.
- (2) **Constructor:** The BakedItem class must contain a constructor that accepts four parameters representing the values to be assigned to the *instance* fields: name, flavor, quantity, and ingredients. The last parameter (e.g., ingredientIn) should be a variable length parameter (i.e., String...ingredientIn), which will be type String[] in the constructor body. Since this class is abstract, the constructor will be called from the subclasses of BakedItem using *super* and the parameter list. The count field should be incremented in this constructor.
- (3) **Methods:** Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
 - o getName: Accepts no parameters and returns a String representing the name.
 - o setName: Accepts a String representing the name, sets the field, and returns nothing.
 - o getFlavor: Accepts no parameters and returns a String representing the flavor.
 - o setFlavor: Accepts a String representing the flavor, sets the field, and returns nothing.
 - o getQuantity: Accepts no parameters and returns a int representing quantity.
 - o setQuantity: Accepts an int representing the quantity, sets the field, and returns nothing.
 - o getIngredients: Accepts no parameters and returns a String[] representing ingredients.
 - o setIngredients: Accepts a variable length parameter (i.e., String...ingredientIn) representing the ingredients, which will be type String[] in the method body, sets the field, and returns nothing.
 - o getCount: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.

- `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
- `toString`: Returns a String describing the BakedItem object. This method will be inherited by the subclasses and should be declared *public final* so that it cannot be overridden. The double value for the price() should be formatted (" \$#,##0.00"), but the numeric value for quantity does not require formatting. Below is an example of the toString result when called on a Cookie object of the Cookie class described below. Note that "Cookie" is the class name which is followed by name and flavor separated by a dash. Quantity and Price are preceded by three spaces. The ingredients, which are enclosed in parentheses, begin on a new line and are separated by commas. Also, there should be at most five ingredients on a line. For example, "salt" is the sixth ingredient so it was printed on the next line.

```
Cookie: Chips Delight - Chocolate Chip   Quantity: 12   Price: $4.20
(Ingredients: flour, sugar, dark chocolate chips, butter, baking soda,
salt)
```

Note that you can get the class name for the instance by calling `this.getClass().toString().substring(6)`. For the example Cookie c, `this.getClass().toString()` returns "class Cookie" and `substring(6)` extracts the class name "Cookie" which begins at character 6. This approach allows the toString method in BakedItem to work for all subclasses that inherit the toString method.

- `price`: An *abstract* method that accepts no parameters and returns a double representing the price for a BakedItem. Since this is an abstract method, it has no body in BakedItem; however, each non-abstract subclass of BakedItem must implement this method.

Code and Test: Since BakedItem is *abstract*, you cannot create instances to test. You will need to use instances of a subclass, e.g., Cookie, which specified below. Thus, it is common to test the methods in an abstract class in the test file for the first non-abstract subclass (i.e., Cookie).

- **Cookie.java**

Requirements: Derive the class Cookie from BakedItem.

Design: The Cookie class has fields, a constructor, and methods as outlined below.

- (1) **Fields:** a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public*, *static*, and *final* modifiers and initialized to 0.35. This is the only field that should be declared in this class.
- (2) **Constructor:** The Cookie class must contain a constructor that accepts four parameters representing the four instance fields in the BakedItem class (name, flavor, quantity, and ingredients). Since this class is a subclass of BakedItem, the super constructor should be called with field values for BakedItem. Below is an example of how the constructor could be used to create a Cookie object:

```
Cookie c = new Cookie("Chips Delight", "Chocolate Chip", 12,
                      "flour", "sugar", "dark chocolate chips",
                      "butter", "baking soda", "salt");
```

(3) **Methods:** Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods.

You will need to implement the following price method but not the toString method.

- o `price`: Accepts no parameters and returns a double representing the price for the Cookie calculated as follows: `BASE_RATE * quantity`.
- o `toString`: None - this method is inherited from `BakedItem`. An example of the return value is shown below for Cookie `c` created above. Note that Quantity and Price are preceded by three spaces.

```
Cookie: Chips Delight - Chocolate Chip   Quantity: 12   Price: $4.20
(Ingredients: flour, sugar, dark chocolate chips, butter, baking soda,
salt)
```

Code and Test: As you implement the Cookie class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. You can now continue developing the methods in `BakedItem` (parent class of `Cookie`). The test methods in `CookieTest` should be used to test the methods in both `BakedItem` and `Cookie`. Remember, *Cookie is-a BakedItem* which means `Cookie` inherited the instance methods defined in `BakedItem`. Therefore, you can create instances of `Cookie` to test methods of the `BakedItem` class.

- **Pie.java**

Requirements: Derive the class `Pie` from `BakedItem`.

Design: The `Pie` class has a field, a constructor, and methods as outlined below.

(1) **Field:** an *instance* variable for crust cost of type double, which is declared with the *private* access modifier; a *class* variable (a constant) `BASE_RATE` of type double, which is declared with the *public*, *static*, and *final* modifiers and initialized to 12.0. These are the only fields that should be declared in this class.

(2) **Constructor:** The `Pie` class must contain a constructor that accepts five parameters representing the four instance fields in the `BakedItem` class and the one instance field crust cost declared in `Pie` (name, flavor, quantity, crust cost, and ingredients). Since this class is a subclass of `BakedItem`, the super constructor should be called with field values for `BakedItem`. Below are examples of how the constructor could be used to create a `Pie p1` (without a crust cost) and `Pie p2` (with a crust cost of \$2).

```
Pie p1 = new Pie("Weekly Special", "Apple", 1, 0,
                "flour", "sugar", "apples", "cinnamon",
                "butter", "baking soda", "salt");

Pie p2 = new Pie("Summer Special", "Key Lime", 1, 2.0,
                "flour", "sugar", "lime juice", "lemon juice",
                "graham crackers", "butter", "baking soda", "salt");
```

(3) **Methods:** Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following `getCrustCost`, `setCrustCost`, and `price` methods but not the `toString` method.

- `getCrustCost`: Accepts no parameters and returns a double representing crust cost.
- `setCrustCost`: Accepts double for crust cost, sets the field, and returns nothing.
- `price`: Accepts no parameters and returns a double representing the price for a Pie calculated as follows: $(\text{BASE_RATE} + \text{crustCost}) * \text{quantity}$.
- `toString`: NONE. When `toString` is invoked on an instance of `Pie`, the `toString` method inherited from `BakedItem` is called. Below is an example of the `toString` result for `Pie p1` as it is declared above.

```
Pie: Weekly Special - Apple    Quantity: 1    Price: $12.00  
(Ingredients: flour, sugar, apples, cinnamon, butter,  
baking soda, salt)
```

```
Pie: Summer Special - Key Lime  Quantity: 1    Price: $14.00  
(Ingredients: flour, sugar, lime juice, lemon juice, graham crackers,  
butter, baking soda, salt)
```

Code and Test: As you implement the `Pie` class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of `Pie` in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding `PieTest.java` file.

- **Cake.java**

Requirements: Derive the class `Cake` from class `BakedItem`.

Design: The `Cake` class has a field, a constructor, and methods as outlined below.

(1) **Field:** *instance* variable for layers of type `int`, which should be declared with the *protected* access modifier; a *class* variable (a constant) `BASE_RATE` of type `double`, which is declared with the *public*, *static*, and *final* modifiers and initialized to 8. These are the only fields that should be declared in this class.

(2) **Constructor:** The `Cake` class must contain a constructor that accepts five parameters representing the four instance fields in the `BakedItem` class and the one instance field layers declared in `Cake` (name, flavor, quantity, layers, and ingredients). Since this class is a subclass of `BakedItem`, the super constructor should be called with field values for `BakedItem`. Below is an example of how the constructor could be used to create a `Cake c1` with one layer and `Cake c2` with two layers.

```
Cake c1 = new Cake("Birthday", "Chocolate", 1, 1,
    "flour", "sugar", "cocoa powder", "vanilla", "eggs",
    "butter", "baking soda", "baking powder", "salt");

Cake c2 = new Cake("2-Layer", "Red Velvet", 1, 2,
    "flour", "sugar", "cocoa powder", "food coloring",
    "eggs", "butter", "baking soda", "baking powder",
    "salt");
```

- (3) **Methods:** Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following getLayers, setLayers, and price methods but not the toString method.
- o getLayers: Accepts no parameters and returns an int representing layers.
 - o setLayers: Accepts an int representing layers, sets the field, and returns nothing.
 - o price: Accepts no parameters and returns a double representing the price for the Cake calculated as follows: (BASE_RATE * layers) * quantity.
 - o toString: None - this class uses the toString method inherited from BakedItem. An example of the return value is shown below for Cake c1 and Cake c2 created above. Note that Quantity and Price are preceded by three spaces.

```
Cake: Birthday - Chocolate    Quantity: 1    Price: $8.00
(Ingredients: flour, sugar, cocoa powder, vanilla, eggs,
butter, baking soda, baking powder, salt)
```

```
Cake: 2-Layer - Red Velvet    Quantity: 1    Price: $16.00
(Ingredients: flour, sugar, cocoa powder, food coloring, eggs,
butter, baking soda, baking powder, salt)
```

Code and Test: As you implement the Cake class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Cake in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding CakeTest.java file.

- **WeddingCake.java**

Requirements: Derive the class WeddingCake from Cake.

Design: The WeddingCake class has a fields, a constructor, and methods as outlined below.

- (1) **Fields:** *instance* variables for tiers of type int which should be declared with the *private* access modifier; a *class* variable (a constant) BASE_RATE of type double, which is declared with the *public*, *static*, and *final* modifiers and initialized to 15.0. These are the only fields that should be declared in this class.

- (2) **Constructor:** The WeddingCake class must contain a constructor that accepts six parameters representing the four values for the instance fields in the BakedItem and two for the instance fields declared in Cake and WeddingCake respectively (name, flavor, quantity, layers, tiers, and ingredients). Since this class is a subclass of Cake, the super constructor should be called with five values for Cake. Below is an example of how the constructor could be used to create a WeddingCake object.


```
WeddingCake c3 = new WeddingCake("3-Layer/3-Tier", "Vanilla", 1, 3, 3,
    "flour", "sugar", "buttermilk", "coffee",
    "eggs", "butter", "baking soda", "baking powder",
    "salt");
```

- (3) **Methods:** Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. You will need to implement the following getTiers, setTiers, and price methods but not the toString method.

- o `getTiers`: Accepts no parameters and returns an int representing tiers.
- o `setTiers`: Accepts an int representing tiers, sets the field, and returns nothing.
- o `price`: Accepts no parameters and returns a double representing the price for a WeddingCake calculated as follows: $(\text{BASE_RATE} * \text{layers} * \text{tiers}) * \text{quantity}$.
- o `toString`: None - this class uses the toString method inherited from BakedItem. An example of the return value is shown below for WeddingCake c3 created above. Note that Quantity and Price are preceded by three spaces.

```
WeddingCake: 3-Layer/3-Tier - Vanilla   Quantity: 1   Price: $135.00
(Ingredients: flour, sugar, buttermilk, coffee, eggs,
butter, baking soda, baking powder, salt)
```

Code and Test: As you implement the WeddingCake class, you should compile and test it as methods are created by using a combination of interactions and JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of WeddingCake in interactions. Remember that when you have an instance on the workbench, you can unfold it to see its values. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding WeddingCakeTest.java file.

UML Class Diagram: If you have not already done so, add your classes to the jGRASP project, then generate the UML class diagram by double-clicking  <UML> for the project in the Open Projects window. Once generated, you can use the mouse to select/drag a class to arrange the diagram like the one on page 1. Alternatively, select BakedItem, right-click on it and select Layout > Tree Down, then select Dependents of Selected.