What makes the latter “better”?

A. Less room for error
B. Easier to understand the global structure
C. All of the above
D. None of the above
E. Whatever
Administrative Details

• Lab 1 is done!
  • (You only have 9 more to go.)

• Lab 2
  • You have PRE-LAB to complete before lab
Agenda

• (More) Inheritance
  ○ Overloading & “this”
    • Overwriting & “super”
• Casting
• Association
• Generics
• Wrapper Class
Chop() / Peel() inside eat()? 

- Currently, Cookie Monster cannot eat Apple and Orange if they are not chopped and peeled, respectively.
- What if you wanted to call chop() and peel() inside eat()?
Overloading

```java
public void eat(Cookie something){
    if(something.isEdible()){
        int tempCalories = something.getCalories();
        calories += tempCalories;
        System.out.println("Me eat " + tempCalories +" calories! Om nom nom nom");
    }
}
```
public void eat(Edible something) {
    if (something.isEdible()) {
        int tempCalories = something.getCalories();
        calories += tempCalories;
        System.out.println("Me eat " + tempCalories + " calories! Om nom nom nom");
    }
}
Overloading

• Overloading is useful when the implementation depends on the parameter types, e.g., `String.valueOf()`:

<table>
<thead>
<tr>
<th>Method Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static String</td>
<td><code>valueOf(boolean b)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(char c)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(char[] data)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(char[] data, int offset, int count)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(double d)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(float f)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(int i)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(long l)</code></td>
</tr>
<tr>
<td>static String</td>
<td><code>valueOf(Object obj)</code></td>
</tr>
</tbody>
</table>
this() in overloaded constructors

public class Baby {
    private String name;
    private int age;

    public Baby(String name) {
        this.name = name;
        this.age = 0;
    }

    public Baby(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return name + "(age " + age + ")";
    }
}

public Baby() {
    this("Isak");
}

public Baby(String name) {
    this.name = name;
    this.age = 0;
}

public Baby(String name, int age) {
    this.name = name;
    this.age = age;
}

public String toString() {
    return name + "(age " + age + ")";
}
Agenda

• (More) Inheritance
  • Overloading & “this”
  • Overwriting & “super”
• Casting
• Association
• Generics
• Wrapper Class
public class Apple extends Fruit{
    private boolean isChopped;

    public Apple(int calories){
        super(calories);
        isChopped = false;
    }

    public boolean isEdible(){
        return isChopped;
    }

    public void chop(){
        isChopped = true;
    }
}
public class BossBaby extends Baby{
    private String position;

    public BossBaby(String name, int age){
        this(name, age, "unemployed");
    }
    public BossBaby(String name, int age, String position){
        super(name,age);
        this.position = position;
    }
    public String toString(){
        return super.toString()
        + ": " + position;
    }
}

Overwriting
public class BossBaby extends Baby{
    private String position;

    public BossBaby(String name, int age){
        this(name, age, "unemployed");
    }
    public BossBaby(String name, int age, String position){
        super(name, age);
        this.position = position;
    }
    public String toString(){
        return super.toString() + ":: " + position;
    }
    ...
}
# Access Modifiers

<table>
<thead>
<tr>
<th></th>
<th>Same Class</th>
<th>Class in the Same Package</th>
<th>Any Subclass</th>
<th>Any Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>None (package)</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Again, be as restrictive as possible!
Agenda

- (More) Inheritance
  - Overloading & “this”
  - Overwriting & “super”
- Casting
  - Association
  - Generics
  - Wrapper Class
Casting

- Implicit Casting (widening conversion)
  - byte → short, int, long, float, or double
  - short/char → int, long, float, or double
  - int → long, float, or double
  - long → float, or double
  - float → double

- Subclass to Superclass

- Explicit Casting (narrow conversion)
  - The opposite direction
    e.g., in equals()
Agenda

• (More) Inheritance
  • Overloading & “this”
  • Overwriting & “super”
• Casting
• Association
• Generics
• Wrapper Class
In real life, information is often stored in key-value pairs:

- word - definition
- license plate - car
- country name - president's name
- CTA - bank account
Association Class

• We want a general class that captures the “key → value” relationship.

```java
public class Association {
    protected Object key;
    protected Object value;
    ...
}
```
// Association is part of the structure package
public class Association {
    protected Object key;
    protected Object value;

    public Association (Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public Object getKey() {
        return key;
    }

    public Object getValue() {
        return value;
    }

    public Object setValue(Object value) {
        Object old = this.value;
        this.value = value;
        return old;
    }

    ...
}

Using Association Class

- We can use type casting:

```java
Association a = new Association("cookie", "A cookie is ...");  
String definition = (String) a.getValue(); 
Association b = new Association("Bill", new Integer(97));  
Integer grade = (Integer) b.getValue(); 
```

"generics to the rescue!"
Agenda

• (More) Inheritance
  • Overloading & “this”
  • Overwriting & “super”
• Casting
• Association
• Generics
  • Wrapper Class
Using Generic Data Types

• Instead of casting Objects, Java supports generic (or parameterized) data types (Read Ch 4)
  • Instead of:
    
    ```java
    Association a = new Association("Bill", new Integer(97));
    Integer grade = (Integer) a.getValue();
    ```
  
  • Use:
    
    ```java
    Association<String, Integer> a =
    new Association<String, Integer>("Bill", new Integer(97));
    Integer grade = a.getValue();
    ```

  • Note, types can be nested:
    
    ```java
    Association<String, Association<String, Integer>> a =
    new Association<String, Association<String, Integer>>();
    Association<String, Integer> a2 = a.getValue();
    Integer grade = a2.getValue();
    ```
Association Class (with generics)

// Association is part of the structure package
public class Association {
    protected Object key;
    protected Object value;

    public Association (Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public Object getKey () {
        return key;
    }

    public Object getValue () {
        return value;
    }

    public Object setValue (Object value) {
        Object old = this.value;
        this.value = value;
        return old;
    }

    ...
}
Agenda

• (More) Inheritance
  • Overloading & “this”
  • Overwriting & “super”
• Casting
• Association
• Generics
• Wrapper Class
Wrapper Class

- In Association<K,V>, K and V cannot be a primitive type
- Luckily, Java provides a wrapper class for each primitive type (java.lang package): Boolean, Character, Byte, Short, Integer, Long, Float, Double.
- Useful when primitive types can’t be used

```java
Association<String,Integer> a = new Association<String,Integer>("Bill", new Integer(97));
```

or for type conversion functionality

```java
int num = Integer.parseInt("2");
```