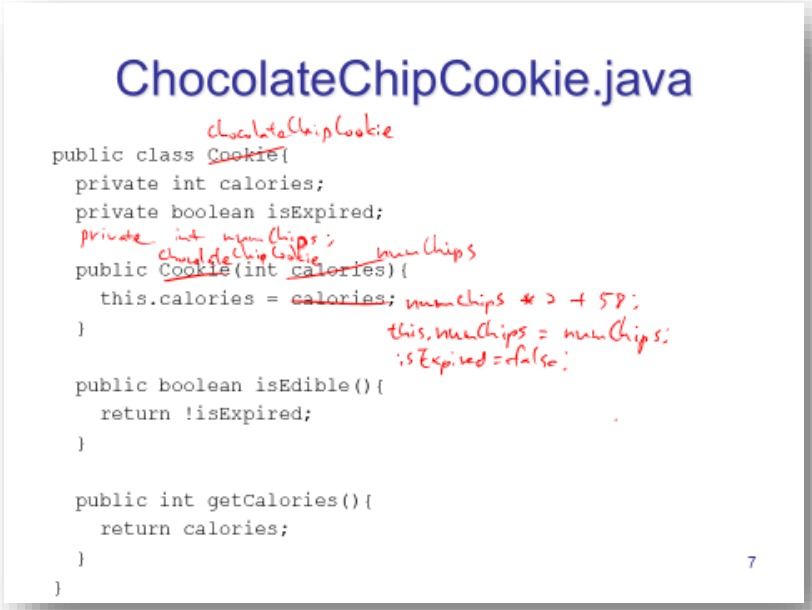
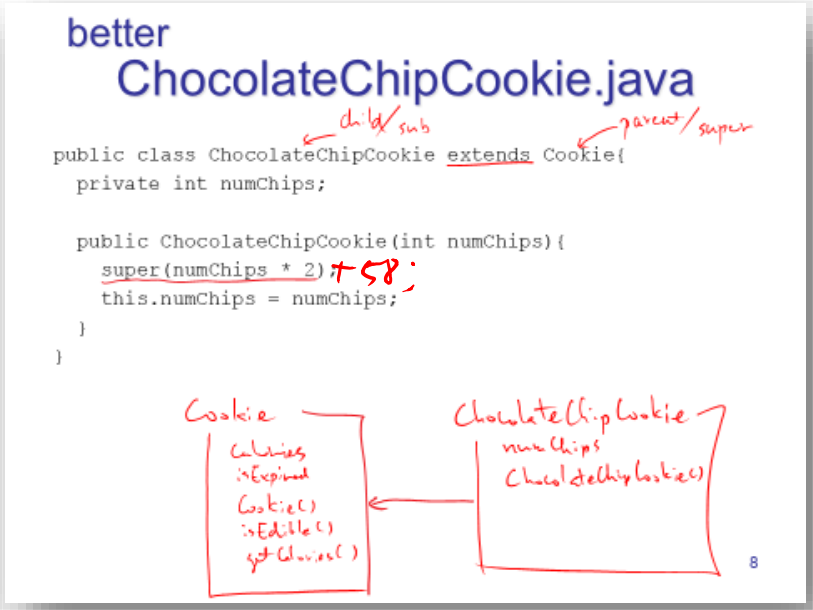
[TAP:AXETF] Inheritance





- 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

better for maintenance

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

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

overloading

Fruit

better CookieMonster.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(Cookie something){
    if(something.isEdible()){
      int tempCalories = something.getCalories();
      calories += tempCalories;
      System.out.println("Me eat " + tempCalories
                  +" calories! Om nom nom nom");
    }
  }
```

Edible

Fruit

CookieMonster.java

```
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");
    }
  }
```

private

eatHelper

```
public void eat(Apple something){
    something.chop();
    eat(something);
}
```

Helper

```
public void eat(Orange something){
    something.peel();
    eat(something);
}
```

Helper

```
public void eat(Edible something){
    eatHelper(something);
}
```

Overloading

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

```
static String   valueOf(boolean b)
static String   valueOf(char c)
static String   valueOf(char[] data)
static String   valueOf(char[] data, int offset, int count)
static String   valueOf(double d)
static String   valueOf(float f)
static String   valueOf(int i)
static String   valueOf(long l)
static String   valueOf(Object obj)
```

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("Jsaki");
}
```

```
this(name, 0);
```

Agenda

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

super()

better Apple.java

```
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;
  }
}
```

Fruit.java

```
public abstract class Fruit{
  private int calories;

  public Fruit(int calories){
    this.calories = calories;
  }

  abstract public boolean isEdible();

  public int getCalories(){
    return calories;
  }
}
```

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;
  }
  …
}
```

overwriting

this() in overloaded c

```
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+")";
  }
}
```

this(name, 0);

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;
  }
  …
}
```

this() in overloaded c

```
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+")";
  }
}
```

Access Modifiers

	Same Class	Class in the Same Package	Any Subclass	Any Class
public	Y	Y	Y	Y
protected	Y	Y	Y	N
None (package)	Y	Y	N	N
private	Y	N	N	N

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

```
int num = 10;
long bigNum = num;
int smallNum = (int) bigNum;
```

e.g. in equals()

Agenda

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

Association

- In real life, information is often stored in key-value pairs:

word – definition

license plate – car

country name – president's name

ISBN – book

CC# – bank account

Association Class

- We want a general class that captures the "key → value" relationship.

```
public class Association {
    protected Object key;
    protected Object value;
    ...
}
```

Association Class

```
// 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:

```
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();
```

String

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:

```
Association a = new Association("Bill",new Integer(97));
Integer grade = (Integer) a.getValue();
```

  - Use:

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

  - Note, types can be nested:

```
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<K,V>{
   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

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

or for type conversion functionality

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