



More About Objects and Methods

Chapter 6

Objectives

- Define and use constructors
- Write and use static variables and methods
- Use methods from class **Math**
- Use predefined wrapper classes
- Use stubs, drivers to test classes and programs

Objectives

- Write and use overloaded methods
- Define and use enumeration methods
- Define and use packages and **import** statements
- Add buttons and icons to applets
- Use event-driven programming in an applet

Constructors: Outline

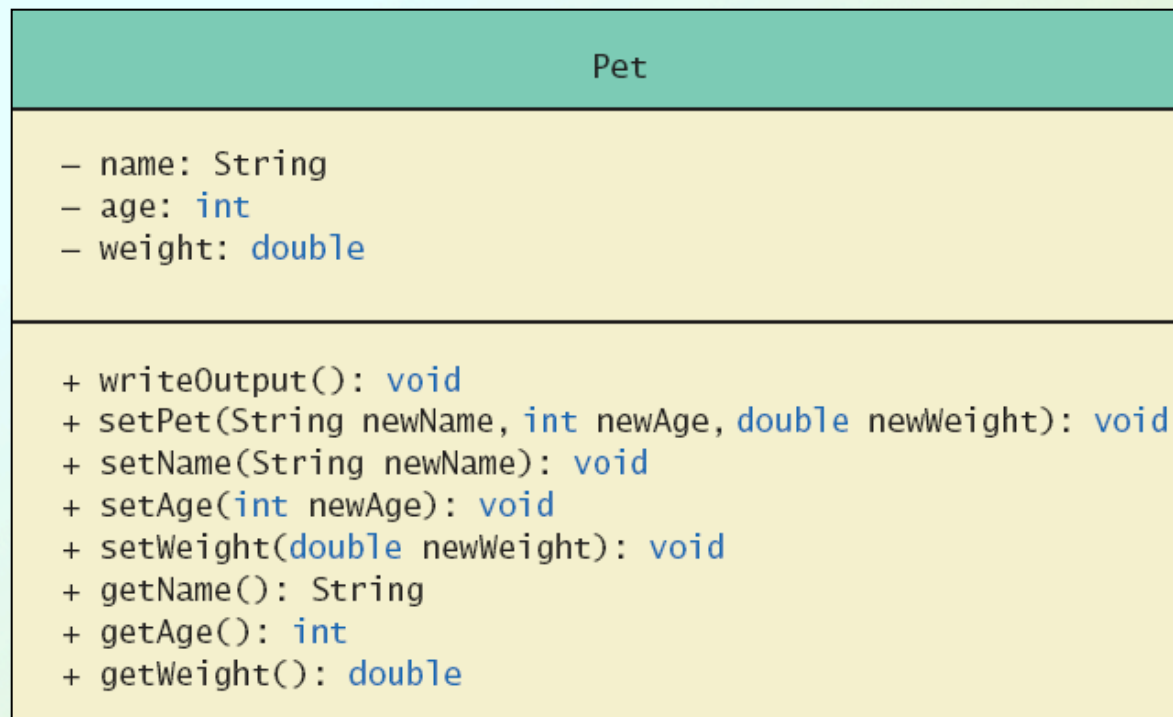
- Defining Constructors
- Calling Methods from Constructors
- Calling a Constructor from Other Constructors

Defining Constructors

- A special method called when instance of an object created with new
 - Create objects
 - Initialize values of instance variables
- Can have parameters
 - To specify initial values if desired
- May have multiple definitions
 - Each with different numbers or types of parameters

Defining Constructors

- Example class to represent pets
- Figure 6.1 Class Diagram for Class **Pet**



Defining Constructors

- Note [sample code](#), listing 6.1
class Pet
- Note different constructors
 - Default
 - With 3 parameters
 - With String parameter
 - With double parameter
- Note [sample program](#), listing 6.2
class PetDemo

Defining Constructors

```
My records on your pet are inaccurate.  
Here is what they currently say:  
Name: Jane Doe  
Age: 0  
Weight: 0.0 pounds  
Please enter the correct pet name:  
Moon Child  
Please enter the correct pet age:  
5  
Please enter the correct pet weight:  
24.5  
My updated records now say:  
Name: Moon Child  
Age: 5  
Weight: 24.5 pounds
```

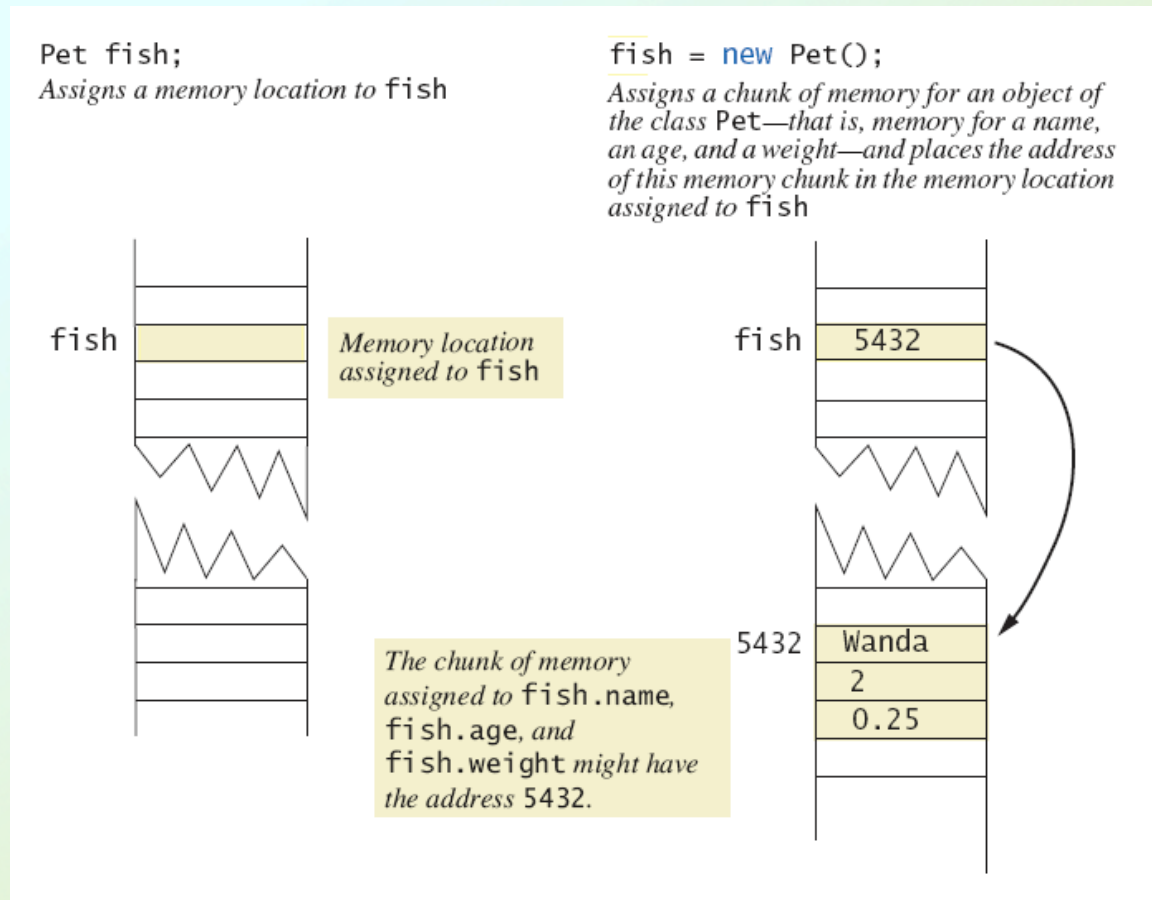
Sample
screen
output

Defining Constructors

- Constructor without parameters is the default constructor
 - Java will define this automatically if the class designer does not define any constructors
 - If you do define a constructor, Java will not automatically define a default constructor
- Usually default constructors not included in class diagram

Defining Constructors


- Figure 6.2 A constructor returning a reference



Calling Methods from Other Constructors

- Constructor can call other class methods

```
public Pet(String initialName, int initialAge,  
           double initialWeight)  
{  
    setPet(initialName, initialAge, initialWeight);  
}
```



- View [sample code](#), listing 6.3

class Pet2

- Note method **set**
- Keeps from repeating code

Calling Constructor from Other Constructors

- From listing 6.3 we have the initial constructor and method set
- In the other constructors use the `this` reference to call initial constructor
- View [revised class](#), listing 6.4
class Pet3
 - Note calls to initial constructor

Static Variables & Methods: Outline

- Static Variables
- Static Methods
- Dividing the Task of a **main** Method into Subtasks
- Adding a **main** Method to a class
- The **Math** Class
- Wrapper Classes

Static Variables

- Static variables are shared by all objects of a class
 - Variables declared **static final** are considered constants – value cannot be changed
- Variables declared **static** (without **final**) can be changed
 - Only one instance of the variable exists
 - It can be accessed by all instances of the class

Static Variables

- Static variables also called *class variables*
 - Contrast with *instance variables*
- Do not confuse class variables with variables of a class type
- Both static variables and instance variables are sometimes called *fields* or *data members*

Static Methods

- Some methods may have no relation to any type of object
- Example
 - Compute max of two integers
 - Convert character from upper- to lower case
- Static method declared in a class
 - Can be invoked without using an object
 - Instead use the class name

Static Methods

- View [sample class](#), listing 6.5
class DimensionConverter
- View [demonstration program](#), listing 6.6
class DimensionConverterDemo

```
Enter a measurement in inches: 18
18.0 inches = 1.5 feet.
Enter a measurement in feet: 1.5
1.5 feet = 18.0 inches.
```

Sample
screen
output

Mixing Static and Nonstatic Methods

- View [sample class](#), listing 6.7
class SavingsAccount
- View [demo program](#), listing 6.8
class SavingsAccountDemo

```
I deposited $10.75.  
You deposited $75.  
You deposited $55.  
You withdrew $15.75.  
You received interest.  
Your savings is $115.3925  
My savings is $10.75  
We opened 2 savings accounts today.
```

Sample
screen
output

Tasks of **main** in Subtasks

- Program may have
 - Complicated logic
 - Repetitive code
- Create static methods to accomplish subtasks
- Consider example code, listing 6.9 a **main** method with repetitive code
- Note alternative code, listing 6.10 uses helping methods

Adding Method **main** to a Class

- Method **main** used so far in its own class within a separate file
- Often useful to include method **main** within class definition
 - To create objects in other classes
 - To be run as a program
- Note [example code](#), listing 6.11 a redefined **class Species**
 - When used as ordinary class, method **main** ignored

The **Math** Class

- Provides many standard mathematical methods
 - Automatically provided, no import needed
- Example methods, figure 6.3a

Name	Description	Argument Type	Return Type	Example	Value Returned
pow	Power	double	double	Math.pow(2.0, 3.0)	8.0
abs	Absolute value	int, long, float, or double	Same as the type of the argument	Math.abs(-7) Math.abs(7) Math.abs(-3.5)	7 7 3.5
max	Maximum	int, long, float, or double	Same as the type of the arguments	Math.max(5, 6) Math.max(5.5, 5.3)	6 5.5

The **Math** Class

- Example methods, figure 6.3b

Name	Description	Argument Type	Return Type	Example	Value Returned
min	Minimum	int, long, float, or double	Same as the type of the arguments	Math.min(5, 6) Math.min(5.5, 5.3)	5 5.3
round	Rounding	float or double	int or long, respectively	Math.round(6.2) Math.round(6.8)	6 7
ceil	Ceiling	double	double	Math.ceil(3.2) Math.ceil(3.9)	4.0 4.0
floor	Floor	double	double	Math.floor(3.2) Math.floor(3.9)	3.0 3.0
sqrt	Square root	double	double	sqrt(4.0)	2.0

Random Numbers

- **Math.random()** returns a random double that is greater than or equal to zero and less than 1
- Java also has a **Random** class to generate random numbers
- Can scale using addition and multiplication; the following simulates rolling a six sided die

```
int die = (int) (6.0 * Math.random()) + 1;
```

Wrapper Classes

- Recall that arguments of primitive type treated differently from those of a class type
 - May need to treat primitive value as an object
- Java provides *wrapper classes* for each primitive type
 - Methods provided to act on values

Wrapper Classes

- Allow programmer to have an object that corresponds to value of primitive type
- Contain useful predefined constants and methods
- Wrapper classes have no default constructor
 - Programmer must specify an initializing value when creating new object
- Wrapper classes have no **set** methods

Wrapper Classes

- Figure 6.4a Static methods in class **Character**

Name	Description	Argument Type	Return Type	Examples	Return Value
toUpperCase	Convert to uppercase	char	char	Character.toUpperCase('a') Character.toUpperCase('A')	'A' 'A'
toLowerCase	Convert to lowercase	char	char	Character.toLowerCase('a') Character.toLowerCase('A')	'a' 'a'
isUpperCase	Test for uppercase	char	boolean	Character.isUpperCase('A') Character.isUpperCase('a')	true false

Wrapper Classes

- Figure 6.4b Static methods in class **Character**

Name	Description	Argument Type	Return Type	Examples	Return Value
isLowerCase	Test for lowercase	char	boolean	Character.isLowerCase('A') Character.isLowerCase('a')	false true
isLetter	Test for a letter	char	boolean	Character.isLetter('A') Character.isLetter('%')	true false
isDigit	Test for a digit	char	boolean	Character.isDigit('5') Character.isDigit('A')	true false
isWhitespace	Test for whitespace	char	boolean	Character.isWhitespace(' ') Character.isWhitespace('A')	true false
Whitespace characters are those that print as white space, such as the blank, the tab character ('\\t'), and the line-break character ('\\n').					

Writing Methods: Outline

- Case Study: Formatting Output
- Decomposition
- Addressing Compiler Concerns
- Testing Methods

Formatting Output

Algorithm to display a double amount as dollars and cents

1. **dollars** = the number of whole dollars in amount.
2. **cents** = the number of cents in amount. Round if there are more than two digits after the decimal point.
3. Display a dollar sign, **dollars**, and a decimal point.
4. Display **cents** as a two-digit integer.



Formatting Output

- View [sample code](#), listing 6.12
class DollarFormatFirstTry
 - Note code to separate dollars and cents
 - Note if-else statement
- View [sample program](#), listing 6.13
class DollarFormatFirstTryDriver
 - Note call to the **write** method

Formatting Output

Testing DollarFormatFirstTry.write:

Enter a value of type double:

1.2345

\$1.23

Test again?

yes

Enter a value of type double:

1.235

\$1.24

Test again?

yes

Enter a value of type double:

9.02

\$9.02

Test again?

yes

Enter a value of type double:

-1.20

\$-1.0-20

Test again?

no

*Oops. There's
a problem here.*

Sample
screen
output

Formatting Output

- View [corrected code](#), listing 6.14
class DollarFormat
 - Note code to handle negative values
- [Program](#) in listing 6.13 will now print values correctly

Decomposition

- Recall pseudocode from [previous slide](#)
- With this pseudocode we *decompose* the task into subtasks
 - Then solve each subtask
 - Combine code of subtasks
 - Place in a method

Addressing Compiler Concerns

- Compiler ensures necessary tasks are done
 - Initialize variables
 - Include **return** statement
- Rule of thumb: believe the compiler
 - Change the code as requested by compiler
 - It is most likely correct

Testing Methods

- To test a method use a driver program
 - Example – code in [listing 6.13](#)
- Every method in a class should be tested
- Bottom-up testing
 - Test code at end of sequence of method calls first
- Use a stub – simplified version of a method for testing purposes

Overloading: Outline

- Overloading Basics
- Overloading and Automatic Type Conversion
- Overloading and the Return Type
- Programming Example: A Class for Money

Overloading Basics

- When two or more methods have same name within the same class
- Java distinguishes the methods by number and types of parameters
 - If it cannot match a call with a definition, it attempts to do type conversions
- A method's name and number and type of parameters is called the *signature*

Overloading Basics

- View [example program](#), listing 6.15
class Overload
- Note overloaded method **getAverage**

```
average1 = 45.0  
average2 = 2.0  
average3 = b
```

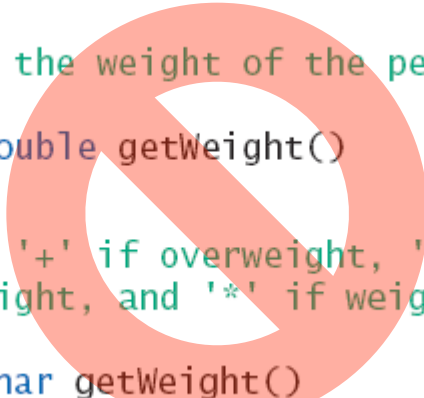
Sample
screen
output

Overloading and Type Conversion

- Overloading and automatic type conversion can conflict
- Recall definition of Pet class of [listing 6.1](#)
 - If we pass an integer to the constructor we get the constructor for age, even if we intended the constructor for weight
- Remember the compiler attempts to overload before it does type conversion
- Use descriptive method names, avoid overloading

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned



```
/**
 * Returns the weight of the pet.
 */
public double getWeight()

/**
 * Returns '+' if overweight, '-' if
 * underweight, and '*' if weight is OK.
 */
public char getWeight()
```

Programming Example

- A class for money
- View [sample class](#), listing 6.16
class Money
- Note use of
 - Private instance variables
 - Methods to set values
 - Methods for doing arithmetic operations

Programming Example

- View [demo program](#), listing 6.17
class MoneyDemo

```
Enter your current savings:  
Enter amount on a line by itself:  
$500.99  
If you double that, you will have $1001.98, or better yet:  
If you triple that original amount, you will have  
$1502.97  
Remember: A penny saved  
is a penny earned.
```

Sample
screen
output

Information Hiding Revisited

Privacy Leaks

- Instance variable of a class type contain address where that object is stored
- Assignment of class variables results in two variables pointing to same object
 - Use of method to change *either* variable, changes the actual object itself
- View [insecure class](#), listing 6.18

class petPair

Information Hiding Revisited

- View [sample program](#),
listing 6.19
class Hacker

```
Our pair:
First pet in the pair:
Name: Faithful Guard Dog
Age: 5 years
Weight: 75.0 pounds
Second pet in the pair:
Name: Loyal Companion
Age: 4 years
Weight: 60.5 pounds

Our pair now:
First pet in the pair:
Name: Dominion Spy
Age: 1200 years
Weight: 500.0 pounds
Second pet in the pair:
Name: Loyal Companion
Age: 4 years
Weight: 60.5 pounds

The pet wasn't so private!
Looks like a security breach.
```

Sample
screen
output

*This program has changed an
object named by a private
instance variable of the object
pair.*

Enumeration as a Class

- Consider defining an enumeration for suits of cards

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

- Compiler creates a class with methods
 - `equals`
 - `compareTo`
 - `ordinal`
 - `toString`
 - `valueOf`

Enumeration as a Class

- View [enhanced enumeration](#), listing 6.20
enum Suit
- Note
 - Instance variables
 - Additional methods
 - Constructor

Packages: Outline

- Packages and Importing
- Package Names and Directories
- Name Clashes

Packages and Importing

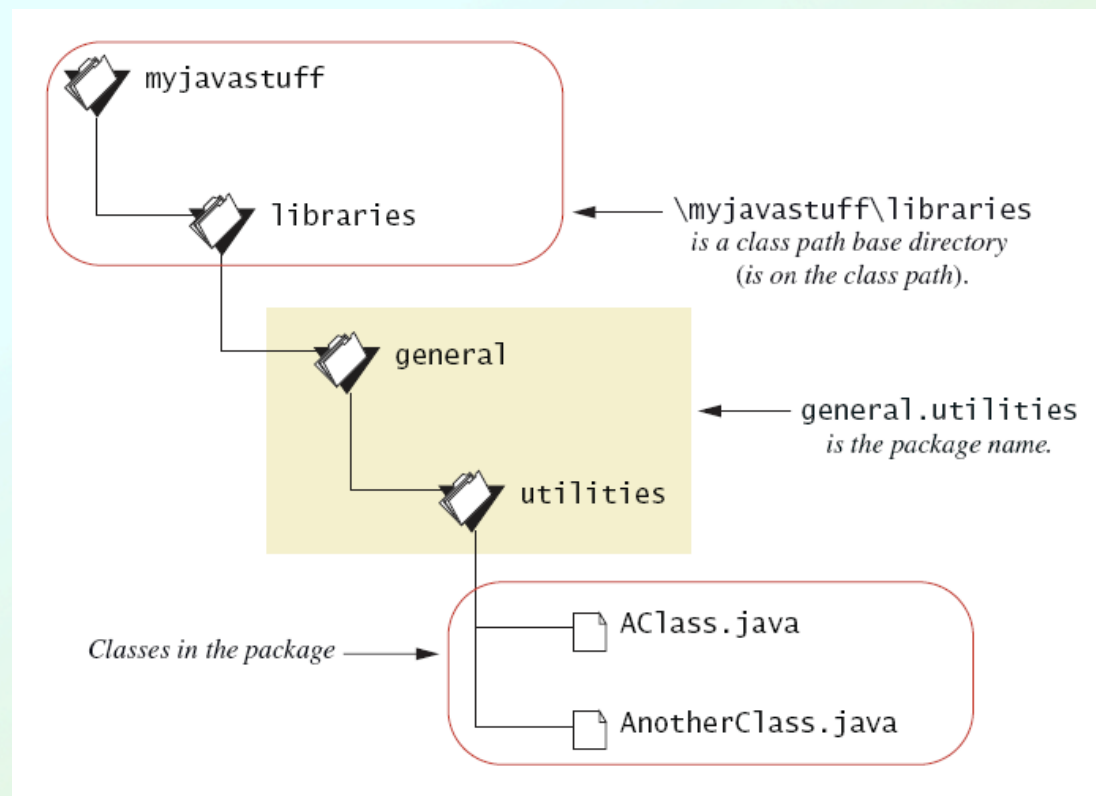
- A package is a collection of classes grouped together into a folder
- Name of folder is name of package
- Each class
 - Placed in a separate file
 - Has this line at the beginning of the file
package **Package_Name**;
- Classes use packages by use of **import** statement

Package Names and Directories

- Package name tells compiler path name for directory containing classes of package
- Search for package begins in class path base directory
 - Package name uses dots in place of / or \
- Name of package uses relative path name starting from any directory in class path

Package Names and Directories

- Figure 6.5 A package name



Name Clashes

- Packages help in dealing with name clashes
 - When two classes have same name
- Different programmers may give same name to two classes
 - Ambiguity resolved by using the package name

Graphics Supplement: Outline

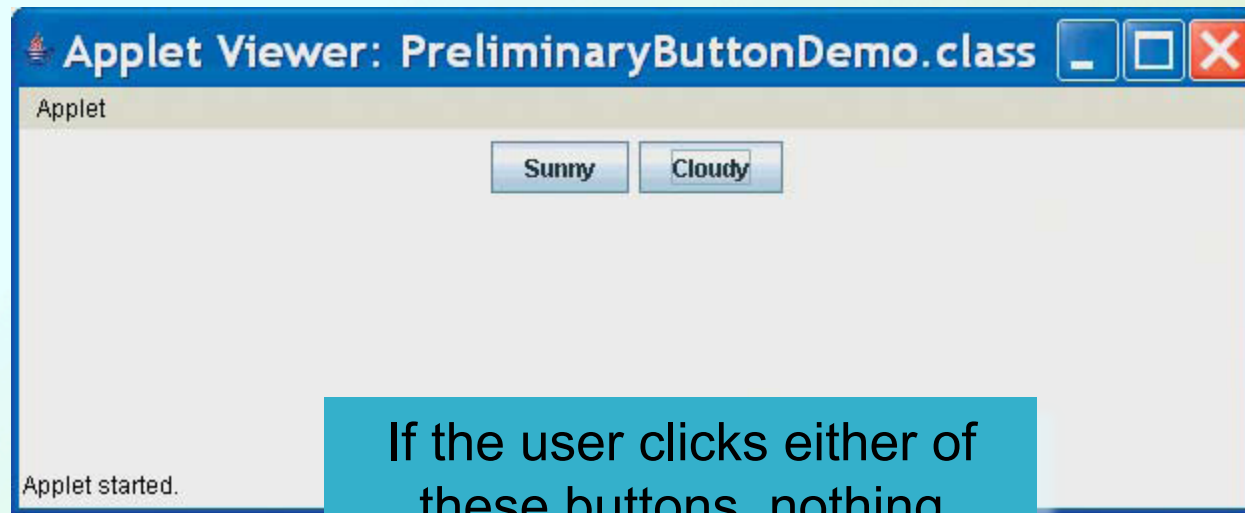
- Adding Buttons
- Event-Driven Programming
- Programming Buttons
- Programming Example: A Complete Applet with Buttons
- Adding Icons
- Changing Visibility
- Programming Example: An Example of Changing Visibility

Adding Buttons

- Create object of type **Jbutton**
 - Then add to content pane
- Possible to associate an action with a button
- View [applet example](#), listing 6.21
class **PreliminaryButtonDemo**

Adding Buttons

- Applet Output



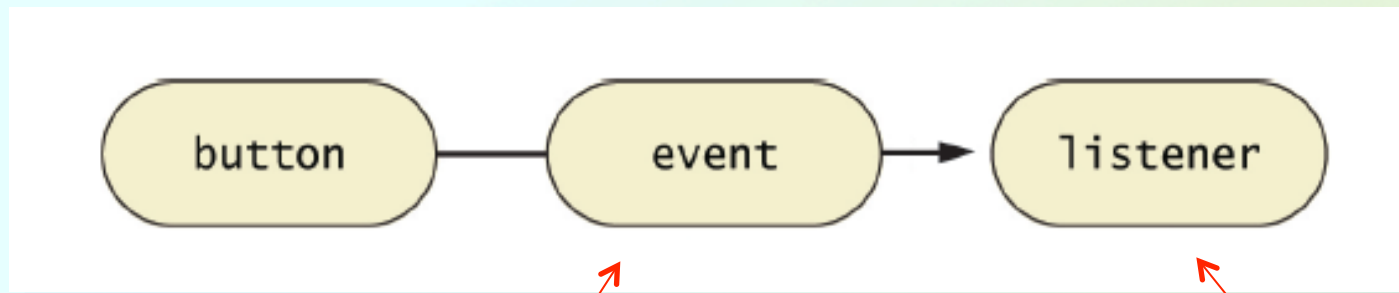
If the user clicks either of these buttons, nothing happens.

Event-Driven Programming

- Applets use events and event handlers
- *An event*
 - An object that represents some user action which elicits a response
 - Example: clicking on button with mouse
- *Listener objects* are specified to receive the events
 - Listener objects have *event handler* methods

Event-Driven Programming

- Figure 6.6 Event firing and an event listener



This event object is the result of a button click. The event object goes from the button to the listener.

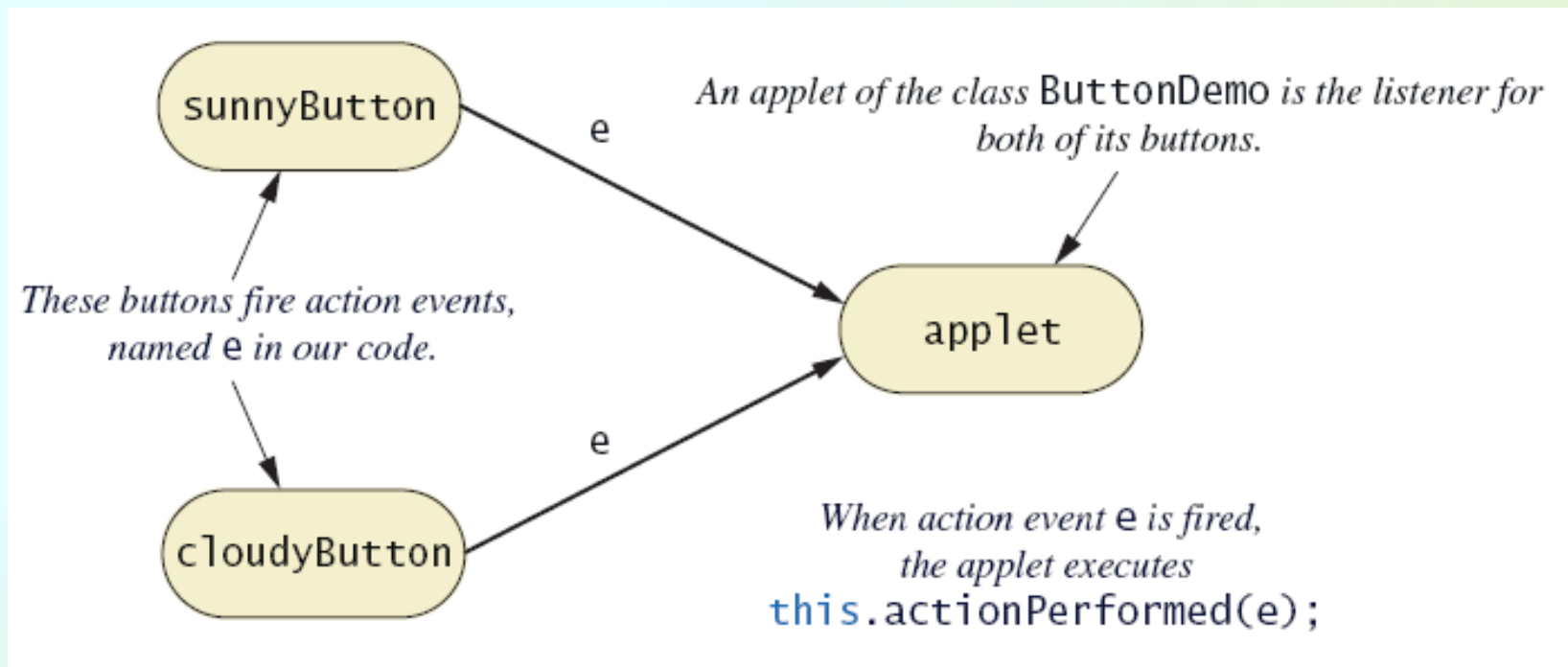
This listener object performs some action, such as making text visible in the applet, when it receives the event object.

Programming Buttons

- When an event is "sent" to the listener object ...
 - A method of the listener object is invoked
 - The event object is given to the listener object method as the argument
- For each button
 - Specify the listener object (register the listener)
 - Methods to be invoked must be defined

Programming Buttons

- Figure 6.7 Buttons and an action listener



Programming Buttons

- Buttons fire events as objects of class **ActionEvent**
- Event objects handled by action listeners
- To make a class an action listener
 - Add phrase **implements ActionListener** to heading of class definition
 - Register the action listener by invoking **addActionListener**
 - Add definition named **actionPerformed** to class

Programming Buttons

- To be an action listener, a class must have
 - A method named **actionPerformed**
 - The method has a parameter of type **ActionEvent**
 - This is the only method required by the **ActionListener** interface
- Syntax

```
public void actionPerformed(ActionEvent e)
{
    Code_for_Actions_Performed
}
```

Programming Example

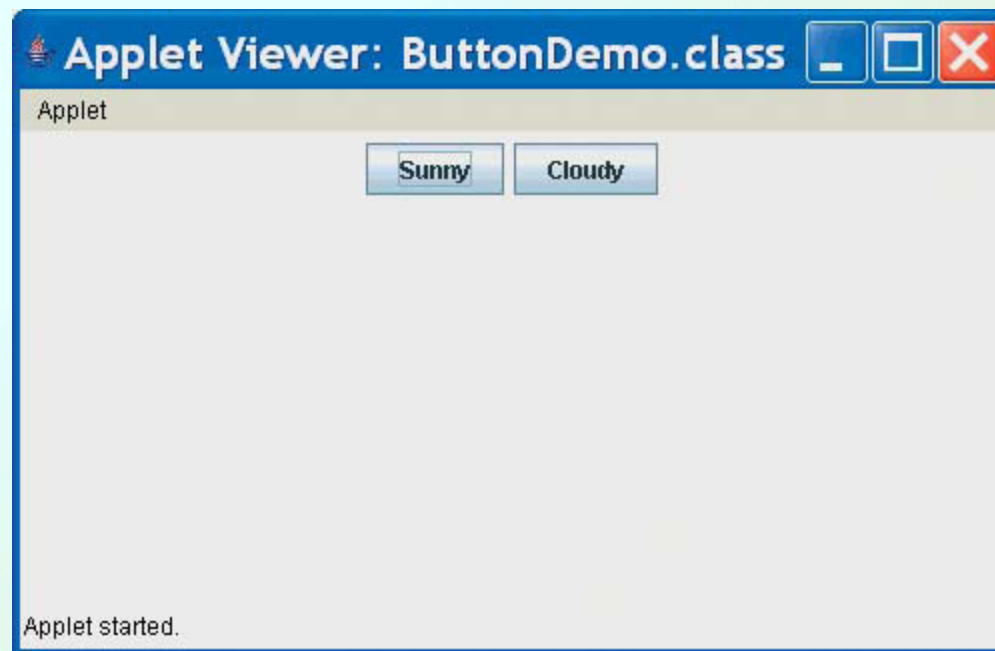
- A Complete Applet with Buttons
- View [applet code](#), listing 6.22

class ButtonDemo

- Note features
 - Specification implements **ActionListener**
 - Invocation of **addActionListener**
 - Method **actionPerformed**

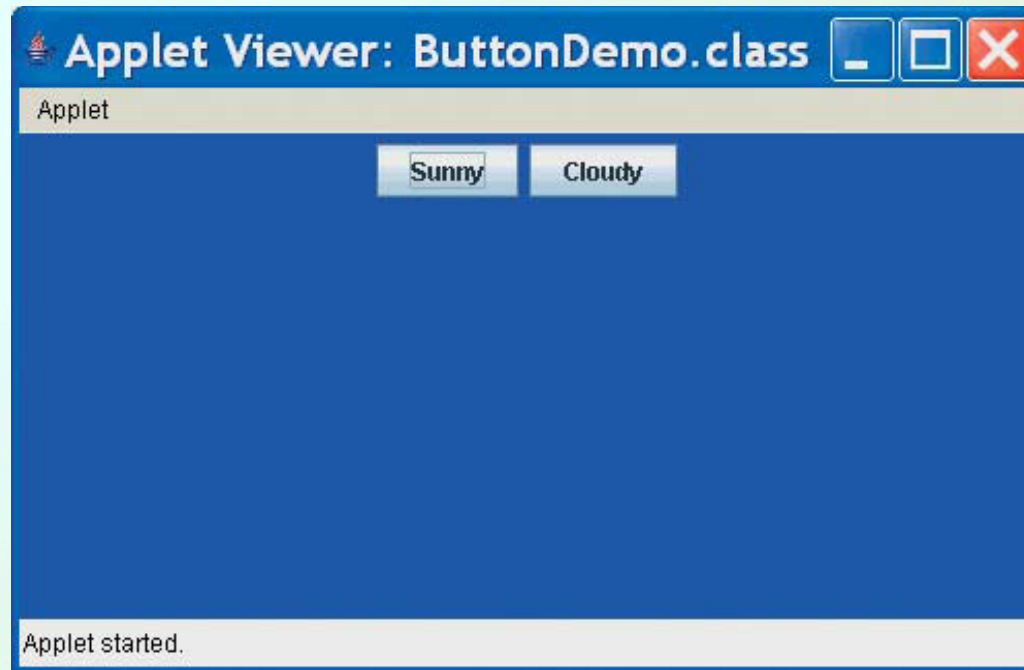
Programming Example

- Initial applet output



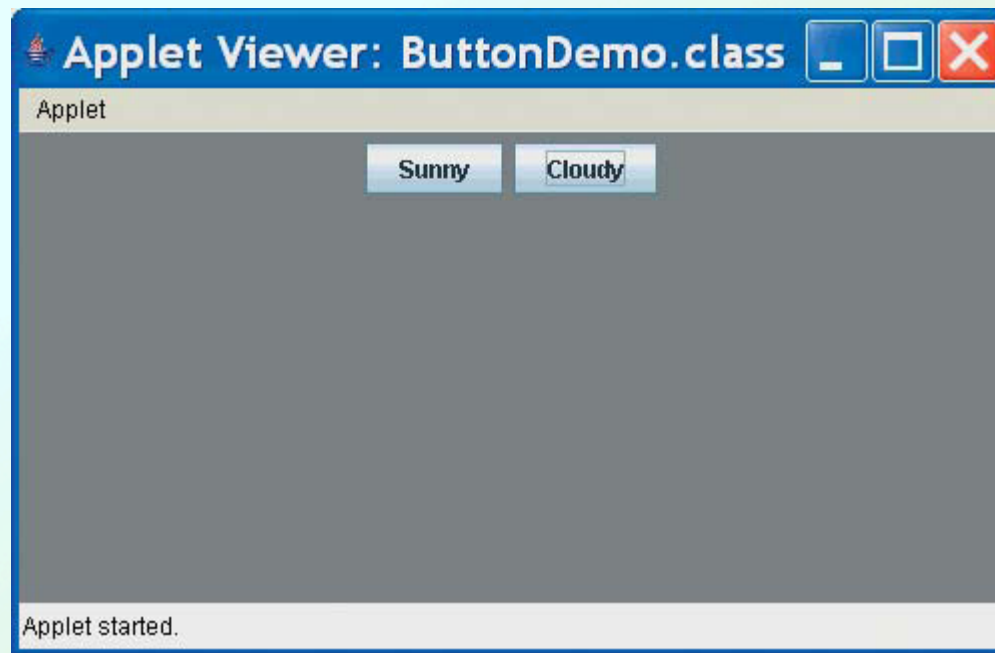
Programming Example

- Applet output after clicking Sunny



Programming Example

- Applet output after clicking Cloudy

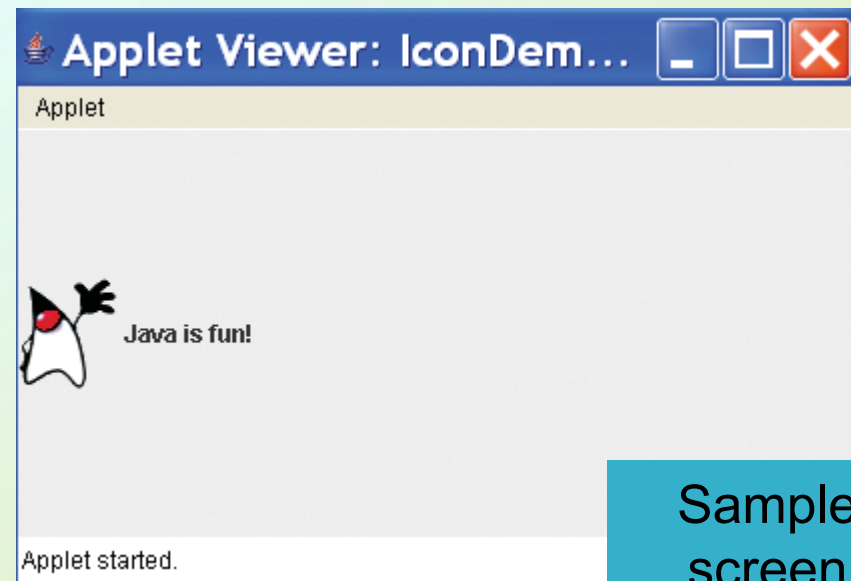


Adding Icons

- An icon is a picture
 - Usually small (but not necessarily)
 - Often a .GIF or .JPEG file
 - Picture file stored in same folder as program
- Icon can be added to a label, button, or other component
- Class **ImageIcon** used to convert digital image to icon

Adding Icons

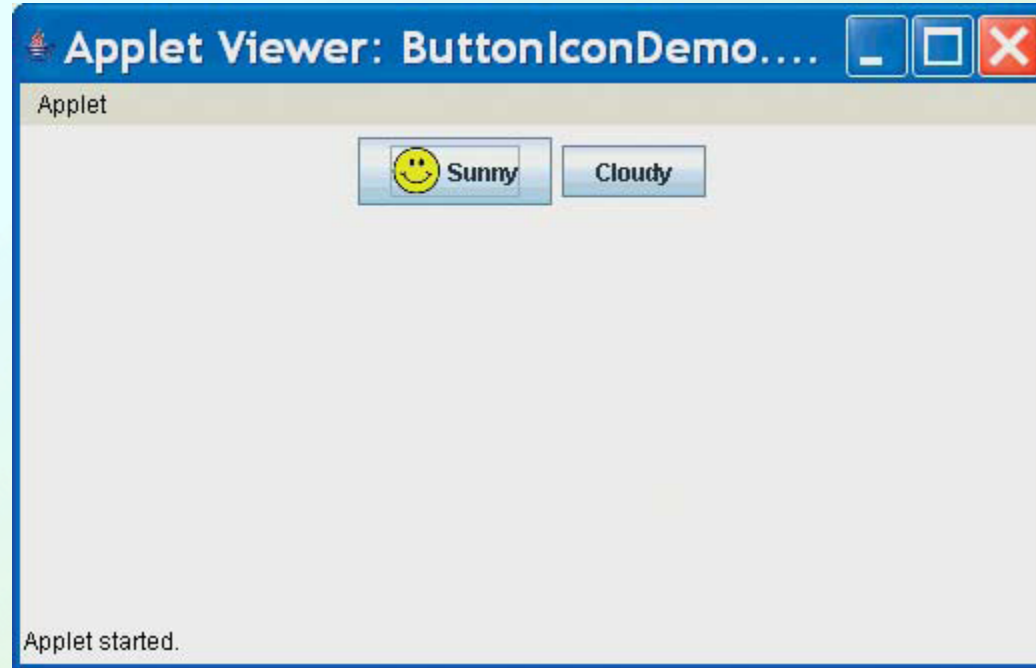
- View [sample applet](#), listing 6.23
class IconDemo
- Note
 - Creation of icon
 - Attaching icon to label



Sample
screen
output

Adding Icons

- Figure 6.8 A button containing an icon



Changing Visibility

- Components have a method named **setVisible**
 - Changes component from visible to invisible (or the other way)
- An invisible component is considered not there
 - Thus an invisible button would be inoperable

Programming Example

- An Example of Changing Visibility
- View [sample applet](#), listing 6.24

class VisibilityDemo



Summary

- Constructor method creates, initializes object of a class
- Default constructor has no parameters
- Within a constructor use this as name for another constructor in same class
- A **static** variable shared by all objects of the class

Summary

- Primitive type has wrapper class to allow treatment as an object
- Java performs automatic type cast between primitive type and object of wrapper class as needed
- Divide method tasks into subtasks
- Test all methods individually

Summary

- Methods with same name, different signatures are overloaded methods
- An enumeration is a class – can have instance variables, constructors, methods
- A package of class definitions grouped together in same folder, contain a **package** statement at beginning of each class