

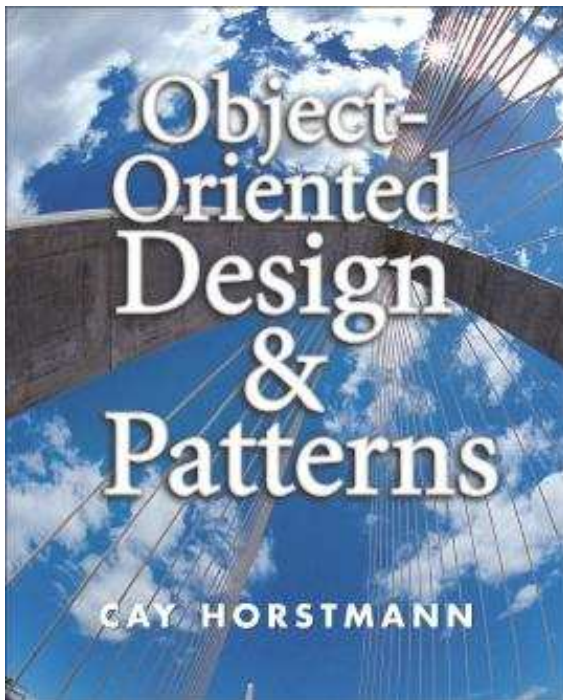
[next](#) ... [Slide 1] ...

Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 1

A Crash Course in Java



[next](#) ... [Slide 1] ...

Chapter Topics

- Hello, World in Java
 - Documentation Comments
 - Primitive Types
 - Control Flow Statements
 - Object References
 - Parameter Passing
 - Packages
 - Basic Exception Handling
 - Strings
 - Reading Input
 - Array Lists and Linked Lists
 - Arrays
 - Static Fields and Methods
 - Programming Style
-

"Hello, World" in Java

- Simple but typical class

Ch1/helloworld/Greeter.java

- Features:
 - Constructor `Greeter(String aName)`
 - Method `sayHello()`
 - Instance field `name`
 - Each feature is tagged `public` or `private`
-

```
01: /**
02:     A class for producing simple greetings.
03: */
04:
05: public class Greeter
06: {
07:     /**
08:         Constructs a Greeter object that can greet a person or
09:         entity.
10:         @param aName the name of the person or entity who should
11:         be addressed in the greetings.
12:     */
13:     public Greeter(String aName)
14:     {
15:         name = aName;
16:     }
17:
18:     /**
19:         Greet with a "Hello" message.
20:         @return a message containing "Hello" and the name of
21:         the greeted person or entity.
22:     */
23:     public String sayHello()
24:     {
25:         return "Hello, " + name + "!";
26:     }
27:
28:     private String name;
29: }
```

"Hello, World" in Java

- Construct new objects with new operator
`new Greeter("World")`
 - Can invoke method on newly constructed object
`new Greeter("World").sayHello()`
 - More common: store object reference in object variable
`Greeter worldGreeter = new Greeter("World");`
 - Then invoke method on variable:
`String greeting = worldGreeter.sayHello();`
-

"Hello, World" in Java

- Construct separate class to test your class

Ch1/helloworld/GreeterTest.java

- `main` method is called when program starts
 - `main` is `static`: it doesn't operate on any objects
 - There are no objects yet when `main` starts
 - In OO program, `main` constructs objects and invokes methods
-

```
1: public class GreeterTest
2: {
3:     public static void main(String[] args)
4:     {
5:         Greeter worldGreeter = new Greeter("World");
6:         String greeting = worldGreeter.sayHello();
7:         System.out.println(greeting);
8:     }
9: }
```

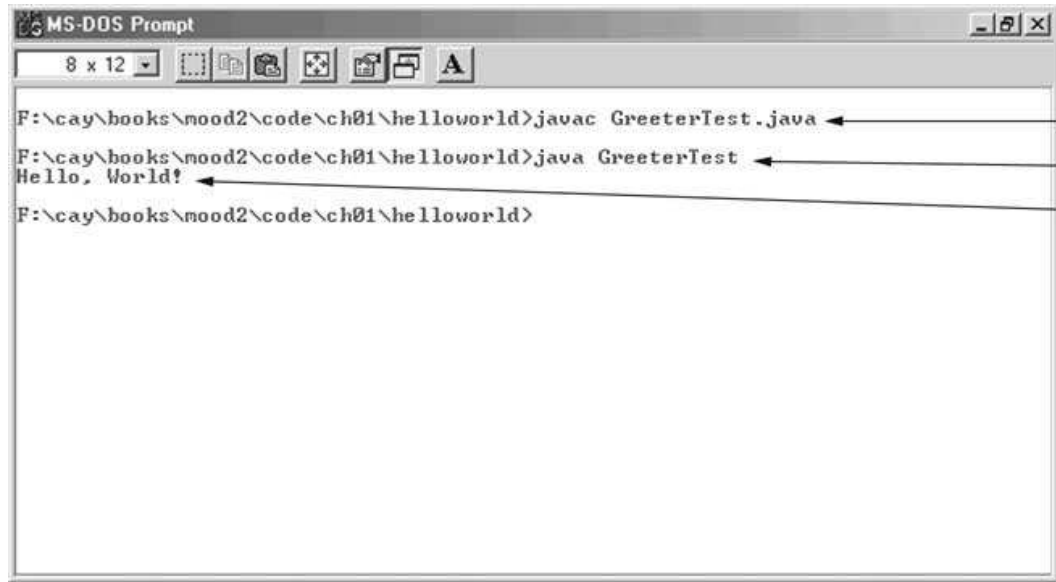
Using the SDK

- Create a new directory to hold your files
 - Use a text editor to prepare files (Greeter.java, GreeterTest.java)
 - Open a shell window
 - cd to directory that holds your files
 - Compile and run

```
javac GreeterTest.java
java GreeterTest
```

Note that Greeter.java is automatically compiled.
 - Output is shown in shell window
-

Using the SDK



The screenshot shows a Windows-style window titled "MS-DOS Prompt" with a standard toolbar. The command prompt displays the following text:

```
F:\cay\books\mood2\code\ch01\helloworld>javac GreeterTest.java  
F:\cay\books\mood2\code\ch01\helloworld>java GreeterTest  
Hello, World!  
F:\cay\books\mood2\code\ch01\helloworld>
```

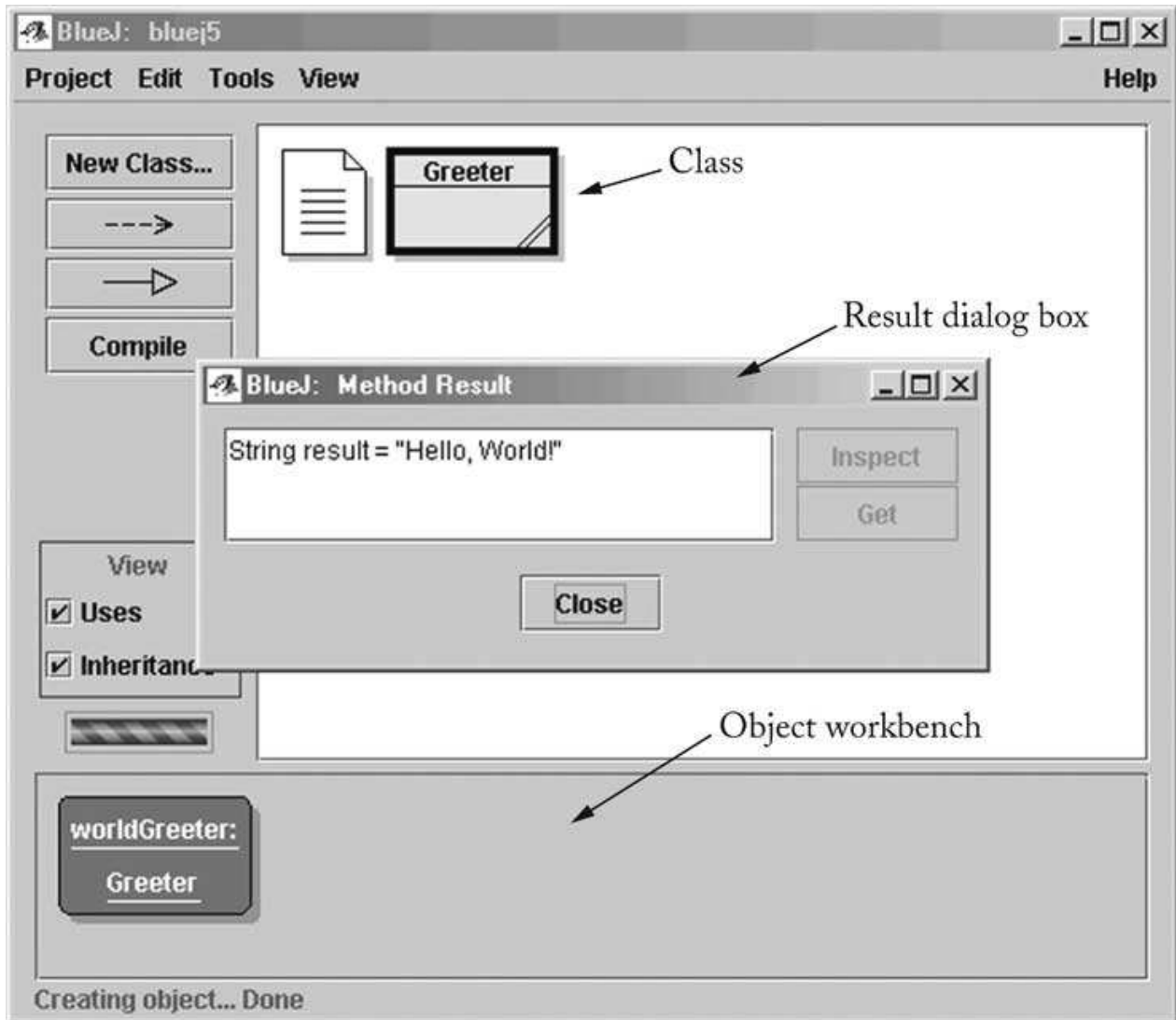
Three arrows point from text labels on the right to the corresponding lines of output:

- "Run compiler" points to the first line: `javac GreeterTest.java`
- "Start interpreter" points to the second line: `java GreeterTest`
- "Message printed" points to the third line: `Hello, World!`

Using BlueJ

- Download BlueJ from <http://www.bluej.org>
 - No test program required
 - Select Project->New and supply directory name
 - Click on New Class... and type in Greeter class
 - Compile the class
 - Right-click the class to construct an object
 - Right-click the object to invoke a method
-

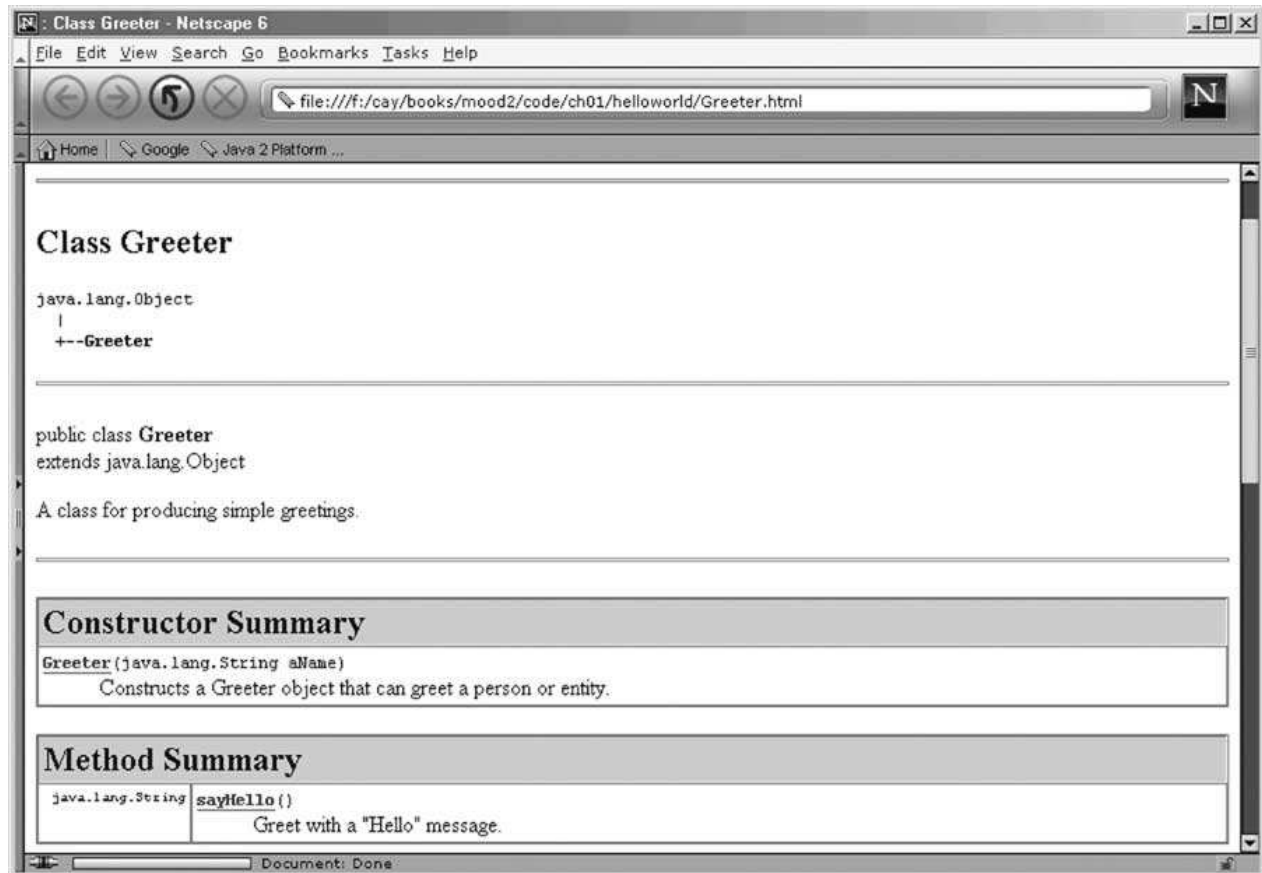
Using BlueJ



Documentation Comments

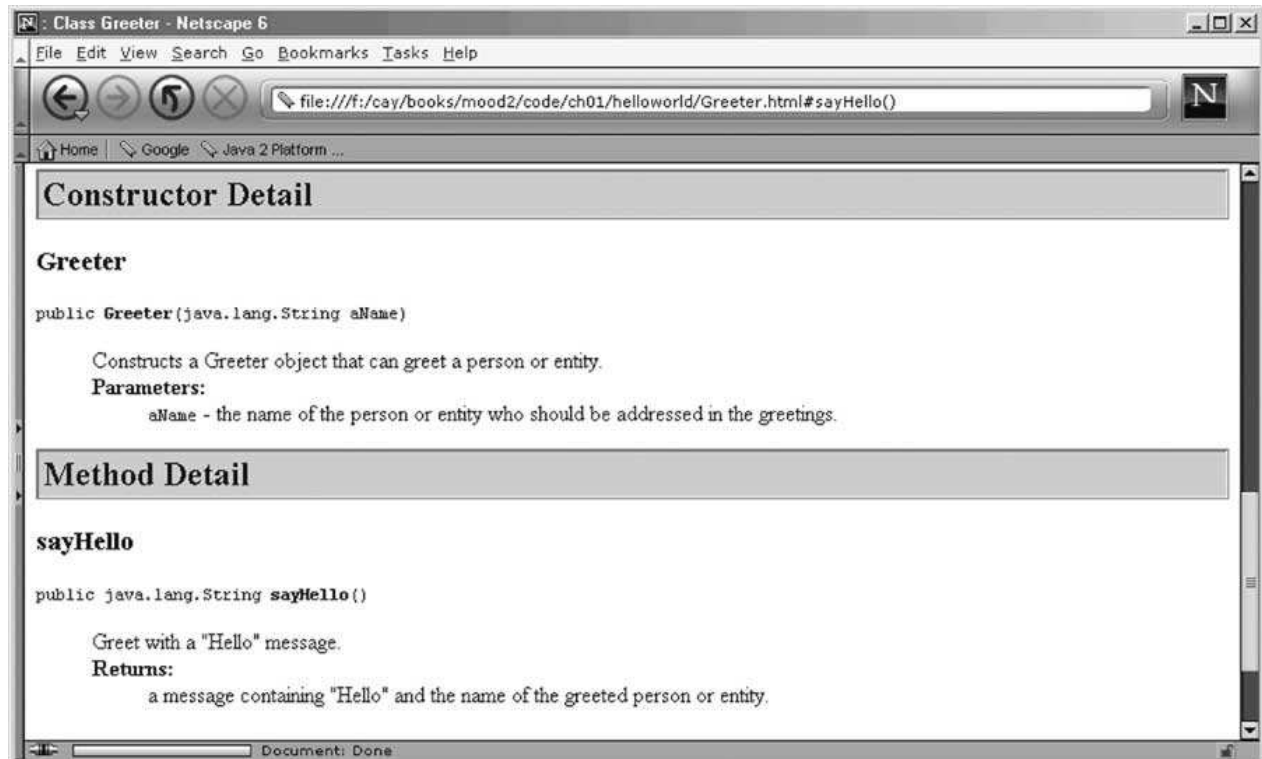
- Delimited by `/** ... */`
 - First sentence = summary
 - `@param` *parameter explanation*
 - `@return` *explanation*
 - Javadoc utility extracts HTML file
-

Documentation Comments - Summary



[previous](#) | [start](#) | [next](#) [Slide 12]

Documentation Comments - Detail

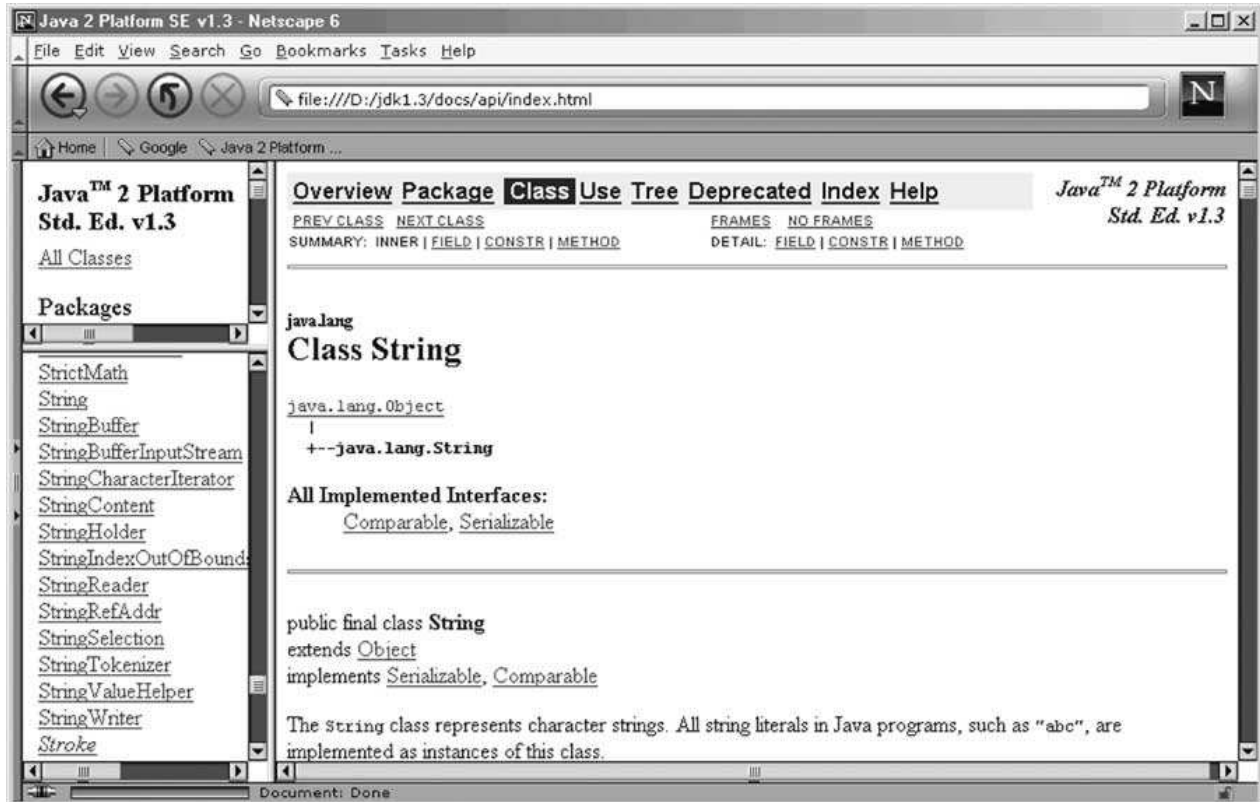


[previous](#) | [start](#) | [next](#) [Slide 12]

Documentation Comments

- Easy to keep documentation in sync with code
 - You must document *all* classes and methods
 - The pros do it--check out the API docs
 - Install and bookmark the API docs *now!*
-

Documentation Comments - API Docs



Primitive Types

- 8 primitive types
 - int, long, short, byte
 - double, float
 - char
 - boolean
 - suffixes L = long, F = float
 - character constants 'a', '\n', '\x2122'
 - Casts (int) x, (float) x
 - Math class has methods that operate on numbers:
`y = Math.sqrt(x);`
-

Control Flow

- if
- while
- do/while
- for
- Variable can be declared in for loop:

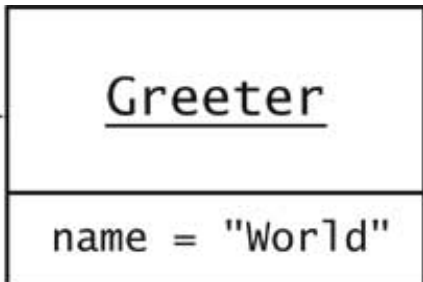
```
for (int i = 1; i <= n; i++)  
{ . . .  
}  
// i no longer defined here
```

Object References

- Object variable holds a *reference*

```
Greeter worldGreeter = new Greeter("World");
```

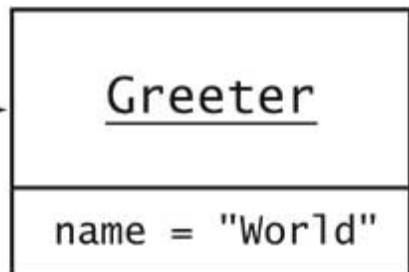
```
worldGreeter =
```



- Can have multiple references to the same object

```
Greeter anotherGreeter = worldGreeter;
```

```
worldGreeter =
```



- After applying mutator method, all references access modified object

```
anotherGreeter.setName("Dave");
```

```
// now worldGreeter.sayHello() returns "Hello,  
Dave!"
```

The null Reference

- null refers to no object
 - Can assign null to object variable:
`worldGreeter = null;`
 - Can test whether reference is null
`if (worldGreeter == null) . . .`
 - Dereferencing null causes `NullPointerException`
-

The `this` Reference

- Refers to implicit parameter of method call
- Example: Equality testing

```
public boolean equals(Greeter other)
{
    if (this == other) return true;
    return name.equals(other.name);
}
```

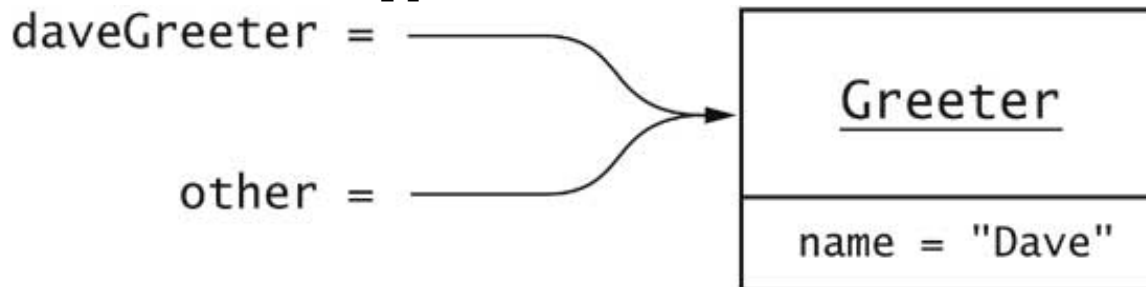
- Example: Constructor

```
public Greeter(String name)
{
    this.name = name;
}
```

Parameter Passing

- Java uses "call by value":
Method receives copy of parameter value
 - Copy of object reference lets method modify object
- ```
public void copyNameTo(Greeter other)
{
 other.name = this.name;
}
```

- ```
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
worldGreeter.copyNameTo(daveGreeter);
```



No Reference Parameters

- Java has no "call by reference"
- ```
public void copyLengthTo(int n)
{
 n = name.length();
}
public void copyGreeterTo(Greeter other)
{
 other = new Greeter(name);
}
```
- Neither call has any effect after the method returns  

```
int length = 0;
worldGreeter.copyLengthTo(length); // length
still 0
worldGreeter.copyGreeterTo(daveGreeter) //
daveGreeter unchanged
```

## Packages

- Classes are grouped into packages
  - Package names are dot-separated identifier sequences
    - `java.util`
    - `javax.swing`
    - `com.sun.misc`
    - `edu.sjsu.cs.cs151.alice`
  - Unique package names: start with reverse domain name
-



## Packages

- Add package statement to top of file  
`package edu.sjsu.cs.cs151.alice;`  
`public class Greeter { . . . }`
  - Class without package name is in "default package"
  - Full name of class = package name + class name  
`java.util.ArrayList`  
`javax.swing.JOptionPane`
-

## Importing Packages

- Tedious to use full class names
  - `import` allows you to use short class name

```
import java.util.ArrayList;
. . .
ArrayList a; // i.e. java.util.ArrayList
```
  - Can import all classes from a package

```
import java.util.*;
```
-

## Importing Packages

- Cannot import from multiple packages  
`import java.*.*; // NO`
  - If a class occurs in two imported packages, import is no help.  
`import java.util.*;  
import java.sql.*;  
...  
java.util.Date d; // Date also occurs in  
java.sql`
  - Never need to import `java.lang`.
-

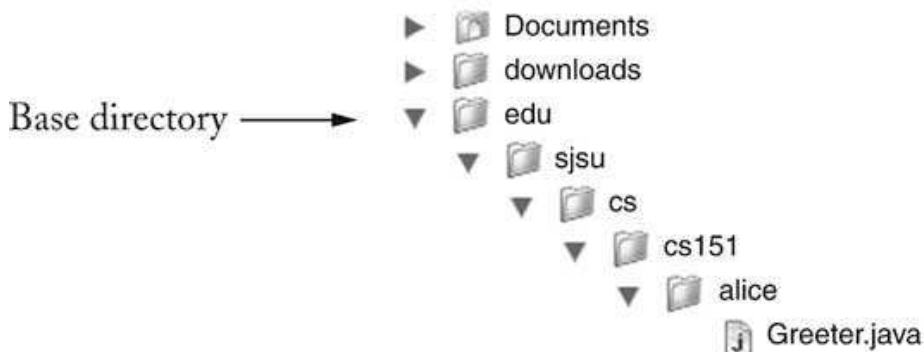
## Packages and Directories

- Package name must match subdirectory name.

`edu.sjsu.cs.sjsu.cs151.alice.Greeter`

must be in subdirectory

`basedirectory/edu/sjsu/cs/sjsu/cs151/alice`



- Always compile from the base directory

```
javac edu/sjsu/cs/sjsu/cs151/alice/Greeter.java
```

or

```
javac edu\sjsu\cs\sjsu\cs151\alice\Greeter.java
```

- Always run from the base directory

```
java edu.sjsu.cs.cs151.alice.GreeterTest
```

---

## Exception Handling

- Example: `NullPointerException`  
`String name = null;`  
`int n = name.length(); // ERROR`
  - Cannot apply a method to `null`
  - Virtual machine *throws* exception
  - Unless there is a handler, program exits with *stack trace*  
`Exception in thread "main"`  
`java.lang.NullPointerException`  
`at Greeter.sayHello(Greeter.java:25)`  
`at GreeterTest.main(GreeterTest.java:6)`
-

## Checked and Unchecked Exceptions

- Compiler tracks only *checked* exceptions
  - `NullPointerException` is not checked
  - `IOException` is checked
  - Generally, checked exceptions are thrown for reasons beyond the programmer's control
  - Two approaches for dealing with checked exceptions
    - Declare the exception in the method header (preferred)
    - Catch the exception
-

## Declaring Checked Exceptions

- Example: Opening a file may throw `FileNotFoundException`:

```
public void read(String filename) throws FileNotFoundException{
 FileReader reader = new FileReader(filename);
 ...
}
```

- Can declare multiple exceptions

```
public void read(String filename)
 throws IOException, ClassNotFoundException

public static void main(String[] args)
 throws IOException, ClassNotFoundException
```

---

## Catching Exceptions

- 

```
try
 { code that might throw an IOException
 }
catch (IOException exception)
 { take corrective action
 }
```

- Corrective action can be:
    - Notify user of error and offer to read another file
    - Log error in error report file
    - In student programs: print stack trace and exit  
`exception.printStackTrace();`  
`System.exit(1);`
-



## The finally Clause

- Cleanup needs to occur during normal and exceptional processing
- Example: Close a file

```
FileReader reader = null;
try
{
 reader = new FileReader(name);
 ...
}
finally
{
 if (reader != null) reader.close();
}
```

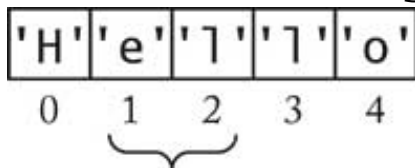
---

## Strings

- Sequence of Unicode characters
  - `length` method yields number of characters
  - `" "` is the empty string of length 0, different from `null`
  - `charAt` method yields characters:  
`char c = s.charAt(i);`
-

## Strings

- substring method yields substrings:  
"Hello".substring(1, 3) is "el"



- Use equals to compare strings  
if (greeting.equals("Hello"))
  - == only tests whether the object references are identical:  
if ("Hello".substring(1, 3) == "el") ... // NO!
-

## StringTokenizer

- Use StringTokenizer to break string into substrings

```
String countries = "Germany,France,Italy";
StringTokenizer tokenizer = new StringTokenizer(countries, ",");
while (tokenizer.hasMoreTokens())
{
 String country = tokenizer.nextToken();
 ...
}
```

---

## String concatenation

- + operator concatenates strings:

```
"Hello, " + name
```

- If one argument of + is a string, the other is converted into a string:

```
int n = 7;
```

```
String greeting = "Hello, " + n;
```

```
// yields "Hello, 7"
```

- toString method is applied to objects

```
Date now = new Date();
```

```
String greeting = "Hello, " + now;
```

```
// concatenates now.toString()
```

```
// yields "Hello, Wed Jan 17 16:57:18 PST 2001"
```

## Converting Strings to Numbers

- Use static methods  
`Integer.parseInt`  
`Double.parseDouble`
  - Example:  

```
String input = "7";
int n = Integer.parseInt(input);
// yields integer 7
```
  - If string doesn't contain a number, throws a `NumberFormatException`(unchecked)
-

## Reading Input

- Use input dialog:  
`String input = JOptionPane.showInputDialog("How old are you?");`
- If user cancels, result is null:  
`if (input != null) age = Integer.parseInt(input);`



## Reading Input

- Must call `System.exit(0)`
- 

### Ch1/input1/InputTest.java

- Also have message dialog  
`JOptionPane.showMessageDialog(null, "Hello, World");`





```
01: import javax.swing.JOptionPane;
02:
03: public class InputTest
04: {
05: public static void main(String[] args)
06: {
07: String input = JOptionPane.showInputDialog("How old are you?");
08: if (input != null)
09: {
10: int age = Integer.parseInt(input);
11: age++;
12: System.out.println("Next year, you'll be " + age);
13: }
14: System.exit(0);
15: }
16: }
```

## Reading Input

- Read console input from System.in
- System.in is an InputStream: reads bytes
- We want a Reader that reads characters
- Turn System.in into InputStreamReader
- Also want to read entire lines
- Use BufferedReader:

```
BufferedReader console = new BufferedReader(
 new InputStreamReader(System.in));
System.out.println("How old are you?");
String input = console.readLine();
int age = Integer.parseInt(input);
```

- 

### [Ch1/input2/InputTest.java](#)

---

```
01: import java.io.BufferedReader;
02: import java.io.IOException;
03: import java.io.InputStreamReader;
04:
05: public class InputTest
06: {
07: public static void main(String[] args)
08: throws IOException
09: {
10: BufferedReader console = new BufferedReader(
11: new InputStreamReader(System.in));
12: System.out.println("How old are you?");
13: String input = console.readLine();
14: if (input != null)
15: {
16: int age = Integer.parseInt(input);
17: age++;
18: System.out.println("Next year, you'll be " + age);
19: }
20: }
21: }
```

## The ArrayList class

- Collects objects of any class type

- add appends to the end

```
ArrayList countries = new ArrayList();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

- size method yields number of elements

```
for (int i = 0; i < countries.size(); i++) . .
.
```

- get gets an element; must cast to correct type:

```
String country = (String)countries.get(i);
```

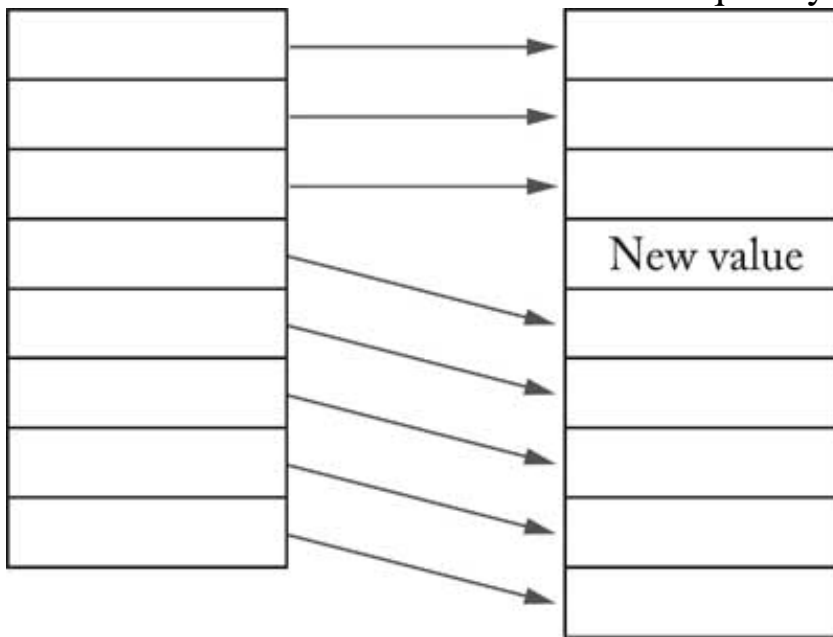
- set sets an element

```
countries.set(1, "France");
```

---

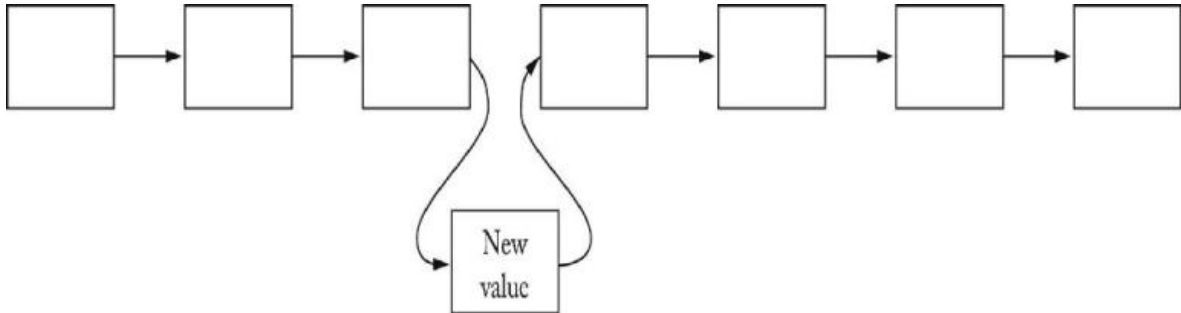
## The ArrayList class

- Insert and remove elements in the middle  
`countries.add(1, "Germany");`  
`countries.remove(0);`
- Not efficient--use linked lists if needed frequently



## Linked Lists

- Efficient insertion and removal



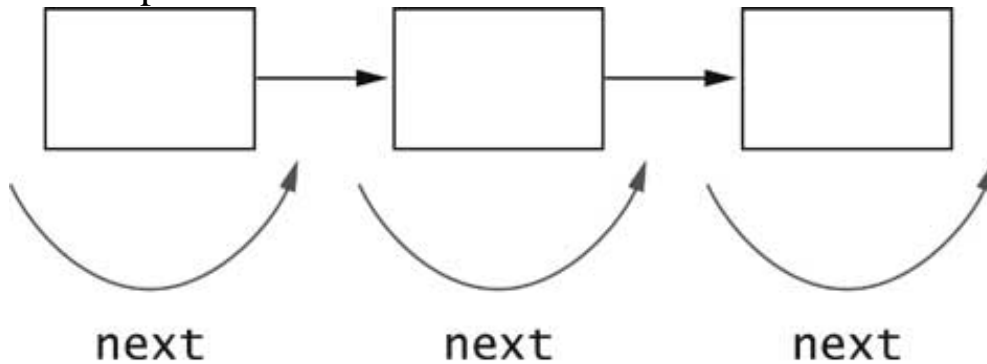
- add appends to the end

```
LinkedList countries = new LinkedList();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

- Use iterators to edit in the middle

## List Iterators

- Iterator points between list elements



- next retrieves element and advances iterator  
`ListIterator iterator = countries.listIterator();`  
`while (iterator.hasNext())`  
`{`  
`String country = (String) iterator.next();`  
`...`  
`}`

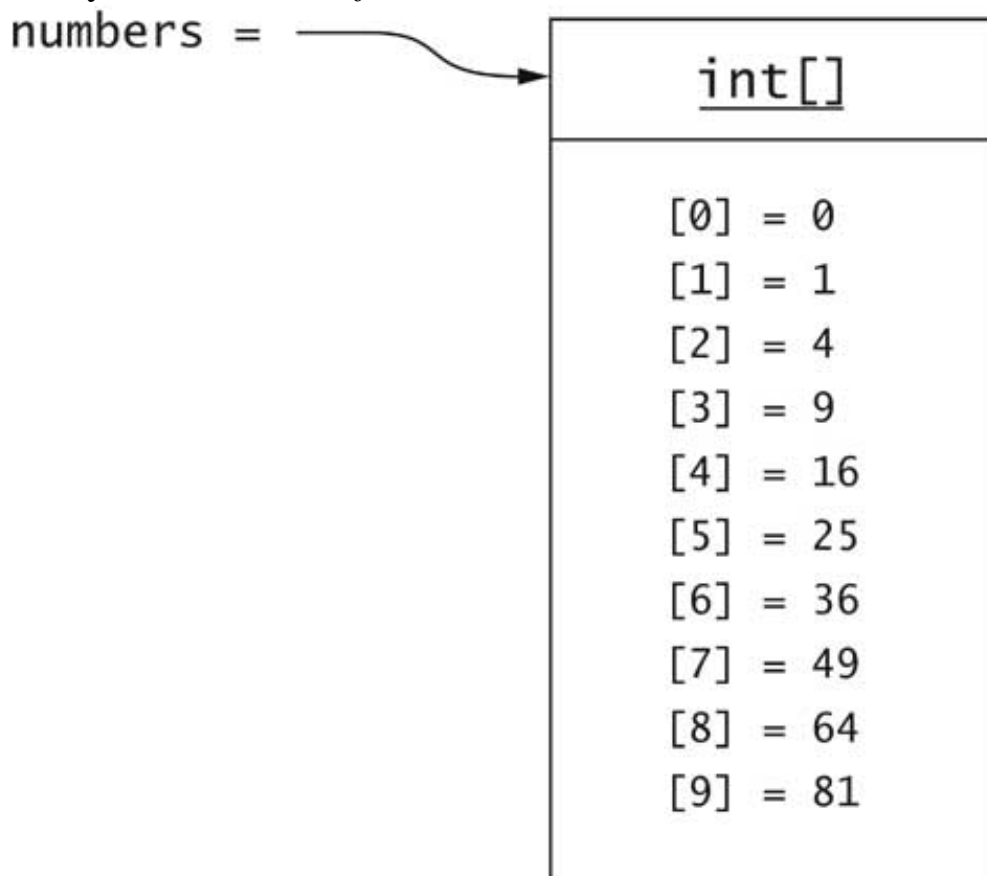
- add adds element before iterator position
  - remove removes element returned by last call to next
-

## Arrays

- Drawback of array lists: can't store numbers
- Arrays can store objects of any type, but their length is fixed

```
int[] numbers = new int[10];
```

- Array variable is a *reference*





## Arrays

- length member yields number of elements  
`for (int i = 0; i < numbers.length; i++)`
  - Can have array of length 0; *not* the same as null:  
`numbers = new int[0];`
  - Array access with [ ] operator:  
`int n = numbers[i];`
  - Multidimensional array  
`int[][] table = new int[10][20];`  
`int t = table[i][j];`
-

## Command-Line Arguments

- `void main(String[] args)`
  - `args` parameter of `main` is initialized with command-line arguments
  - Example:  
`java GreeterTest Mars`
  - `args.length` is 1  
`args[0]` is "Mars"
-

## Static Fields

- Shared among all instances of a class
- Example: shared random number generator

```
public class Greeter
{
 . . .
 private static Random generator;
}
```

- Example: shared constants

```
public class Math
{
 . . .
 public static final double PI =
3.14159265358979323846;
}
```

---

## Static Methods

- Don't operate on objects
- Example: `Math.sqrt`
- Example: *factory method*

```
public static Greeter getRandomInstance()
{ if (generator.nextBoolean()) // note: generator is static field
 return new Greeter("Mars");
 else
 return new Greeter("Venus");
}
```

- Invoke through class:  
`Greeter g = Greeter.getRandomInstance();`
  - Static fields and methods should be rare in OO programs
-

## Programming Style: Case Convention

- variables, fields and methods:  
start with lowercase, use caps for new words:  
name  
sayHello
  - Classes:  
start with uppercase, use caps for new words:  
Greeter  
ArrayList
  - Constants:  
use all caps, underscores to separate words  
PI  
MAX\_VALUE
-

## Programming Style: Property Access

- Common to use get / set prefixes:

```
String getName()
```

```
void setName(String newValue)
```

- Boolean property has is / set prefixes:

```
public boolean isPolite()
```

```
public void setPolite(boolean newValue)
```

---

## Programming Style: Braces

- "Allman" brace style: braces line up

```
public String sayHello()
{
 return "Hello, " + name + "!";
}
```

- "Kernighan and Ritchie" brace style: saves a line

```
public String sayHello() {
 return "Hello, " + name + "!";
}
```

---

## Programming Style: Fields

- Some programmers put fields before methods:

```
public class Greeter
{
 private String name;
 public Greeter(String aName) { . . . }
 . . .
}
```

- From OO perspective, it is better to list the `public` interface first
  - All fields should be `private`
  - Don't use default (package) visibility
-



## Programming Style: Miscellaneous

- Spaces around operators, after keywords, but not after method names

Good: `if (x > Math.sqrt(y))`

Bad: `if(x>Math.sqrt (y))`

- Don't use C-style arrays:

Good: `int[] numbers`

Bad: `int numbers[]`

- No magic numbers

Good: `h = HASH_MULTIPLIER * h + val[off];`

Bad: `h = 31 * h + val[off];`

---