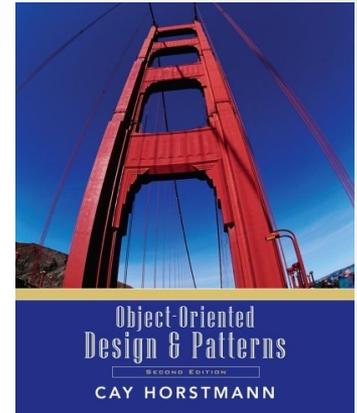


Object-Oriented Design & Patterns

2nd edition
Cay S. Horstmann



Chapter 1: A Crash Course in Java

CPSC 2100

Software Design and Development

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

Chapter Objective

The purpose of this chapter is to teach you the elements of the Java programming language or to give you an opportunity to review them assuming that you know an object-oriented programming language.

“Hello, World!” in Java

Greeter.java

```
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: }
```

- Construct new objects with new operator (Instantiating the class)

```
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

GreeterTester.java

```
1: public class GreeterTester
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: }
```

- **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.

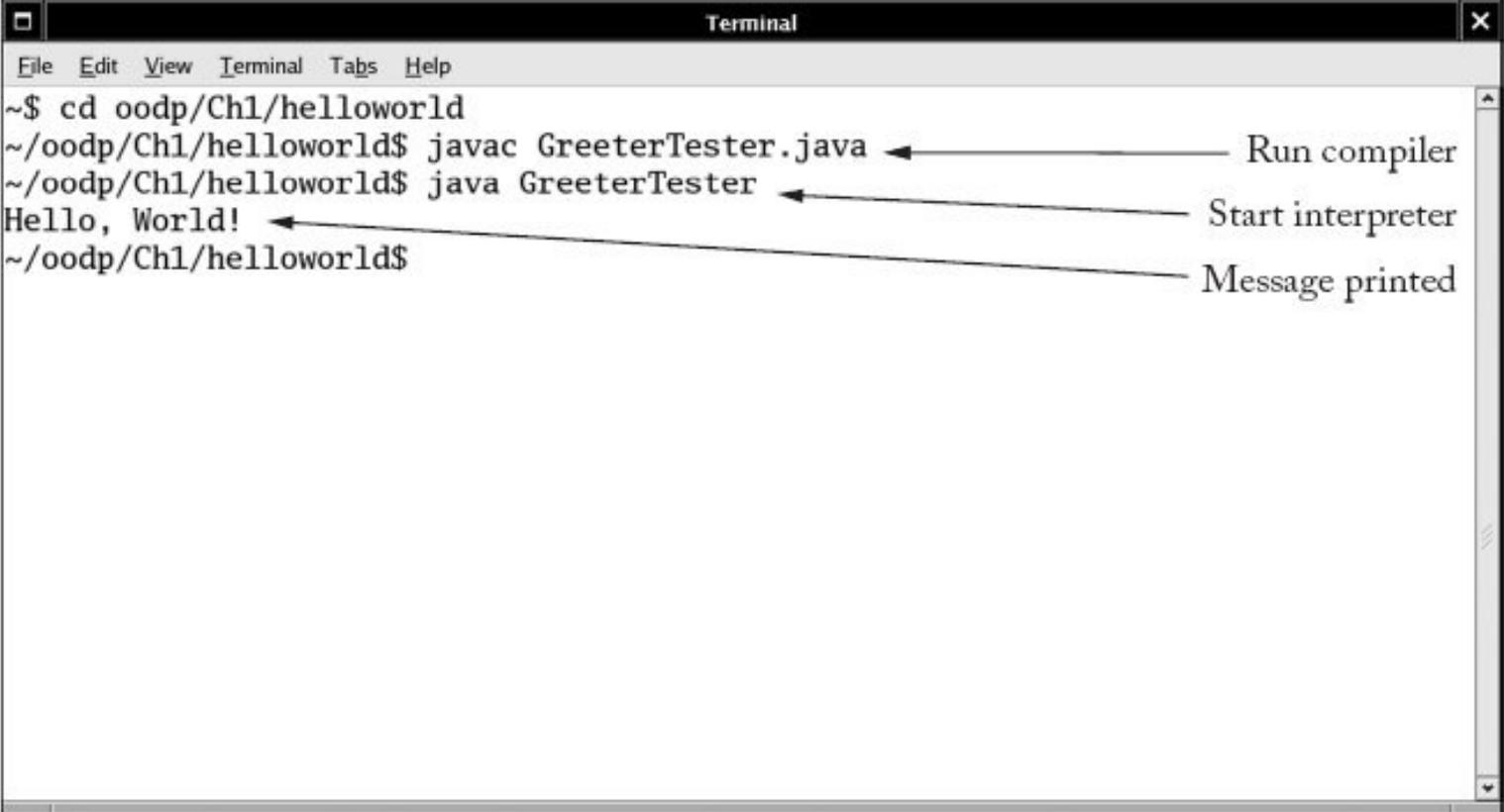
Using the SDK

Software Development Kit

1. Create a new directory to hold your files.
2. Use a text editor to prepare files (Greeter.java, GreeterTest.java).
3. Open a shell window.
4. Use the cd command to the directory that holds your files.
5. **Compile: javac** GreeterTest.java
6. **Run: java** GreeterTest
7. Note that Greeter.java is automatically compiled.
8. Output is shown in shell window

Using the SDK

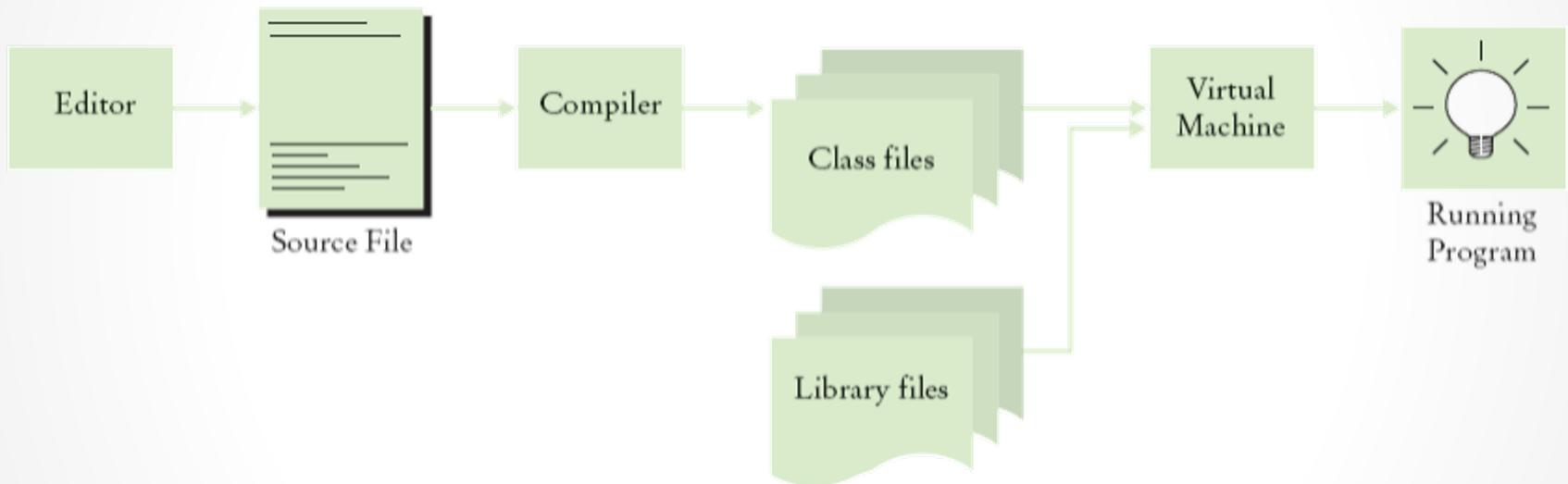
Software Development Kit



```
Terminal
File Edit View Terminal Tabs Help
~$ cd oodp/Ch1/helloworld
~/oodp/Ch1/helloworld$ javac GreeterTester.java
~/oodp/Ch1/helloworld$ java GreeterTester
Hello, World!
~/oodp/Ch1/helloworld$
```

Annotations:

- Run compiler (points to `javac GreeterTester.java`)
- Start interpreter (points to `java GreeterTester`)
- Message printed (points to `Hello, World!`)



Big java, 4th edition, Horstmann, Wiley.

Documentation Comments

- Delimited by `/** ... */`
- First sentence = summary.
- **@param** followed by the parameter name and small explanation.
- **@return** describe the return value.
- **Javadoc** utility extracts HTML file

Whoa! I am supposed to write all this stuff?

- The **Javadoc** utility will format your comments into a nicely formatted set of HTML documents.
- It is possible to spend more time pondering whether a comment is too trivial to write than it takes just to write it.
every class, every method, every parameter, every return value should have a comment.
- It is a good idea to write the method comment first, before writing the method code (Understand what you need to program).

Greeter - Mozilla

File Edit View Go Bookmarks Tools Window Help

file:///home/cay/oodp/Ch1/helloworld/Greeter.html

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS [FRAMES](#) [NO FRAMES](#) [All Classes](#)
SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

Class Greeter

java.lang.Object
 └ Greeter

public class **Greeter**
extends java.lang.Object

A class for producing simple greetings.

Constructor Summary

[Greeter](#)(java.lang.String aName)
Constructs a Greeter object that can greet a person or entity.

Method Summary

| | |
|------------------|-----------------------------|
| java.lang.String | sayHello () |
|------------------|-----------------------------|

Greet with a "Hello" message.

Done

Greeter - Mozilla

File Edit View Go Bookmarks Tools Window Help

file:///home/cay/oodp/Ch1/helloworld/Greeter.html

Constructor Detail

Greeter

public **Greeter**(java.lang.String aName)

Constructs a Greeter object that can greet a person or entity.

Parameters:
aName - the name of the person or entity who should be addressed in the greetings.

Method Detail

sayHello

public java.lang.String **sayHello**()

Greet with a "Hello" message.

Returns:
a message containing "Hello" and the name of the greeted person or entity.

Done

Javadoc utility

- After you have written the documentation comments, invoke the **Javadoc** utility.
 1. Open the shell window.
 2. Use the cd command to change to the directory you have your files in.
 3. Run the Javadoc utility
 4. Javadoc *.java

Check the Java development kit documentation
Application programming interface (API)

<http://docs.oracle.com/javase/7/docs/api/>

Primitive Types

| Type | Size | Range |
|---------|---------|---|
| int | 4 bytes | -2,147,483,648 ... 2,147,483,647 |
| long | 8 bytes | -9,223,372,036,854,775,808L ... 9,223,372,036,854,775,807L |
| short | 2 bytes | -32768 ... 32767 |
| byte | 1 byte | -128 ... 127 |
| char | 2 bytes | '\u0000' '\uFFFF' http://www.unicode.org |
| boolean | | false, true |
| double | 8 bytes | approximatly 1.79769313486231570E+308 |
| float | 4 bytes | approximatly 3.40282347E+38F |

Character Escape Sequence

| Escape Sequence | Meaning |
|---|-----------------------------------|
| <code>\b</code> | backspace (<code>\u0008</code>) |
| <code>\f</code> | form feed (<code>\u000C</code>) |
| <code>\n</code> | newline (<code>\u000A</code>) |
| <code>\r</code> | return (<code>\u000D</code>) |
| <code>\t</code> | tab (<code>\u0009</code>) |
| <code>\\</code> | backslash |
| <code>\'</code> | single quote |
| <code>\"</code> | double quote |
| <code>\un₁n₂n₃n₄</code> | Unicode encoding |

Table 2

Character Escape Sequences

Conversion and Casting

- No Information loss.
 - short to int
 - float to double
- **Casting** is needed to avoid the loss of precision.

All integer types can be converted to float or double, even though some of the conversions (such as long to double) lose precision.

```
double x = 10.0 / 3.0; // sets x to  
3.33333333333333335
```

```
int n = (int) x; //sets n to 3
```

```
float f = (float) x; //sets f to 3.3333333
```

Math Class

- Does not operate on objects.
- Numbers are supplied as parameters.

```
double y = Math.sqrt(x);
```

| Method | Description |
|--------------------------------|--|
| <code>Math.sqrt(x)</code> | Square root of x , \sqrt{x} |
| <code>Math.pow(x, y)</code> | x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer) |
| <code>Math.toRadians(x)</code> | Converts x degrees to radians (i.e., returns $x \cdot \pi/180$) |
| <code>Math.toDegrees(x)</code> | Converts x radians to degrees (i.e., returns $x \cdot 180/\pi$) |
| <code>Math.round(x)</code> | Closest integer to x (as a long) |
| <code>Math.abs(x)</code> | Absolute value $ x $ |

Table 3

Mathematical Methods

Control Flow Statements

If statement:

```
if (x >= 0)
    y = Math.sqrt(x);
else
    y = 0;
```

Control Flow Statements

while statement:

```
while ( x < target)
{
    x = x * a;
    n++;
}
```

do statement:

```
do
{
    x = x * a;
    n++;
} while ( x <
target);
```

Control Flow Statements

for statement:

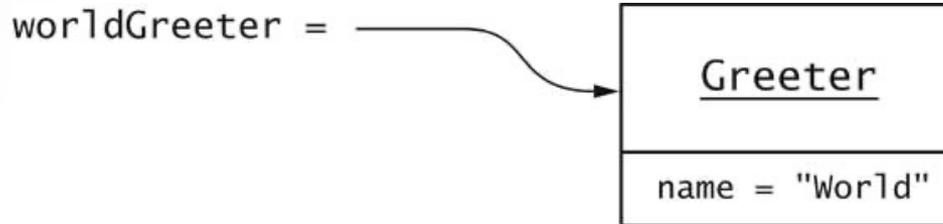
```
for (i = 1; i <= n; i++)  
{  
    x = x * a;  
    sum = sum + x;  
}
```

Variable scope.

Object References

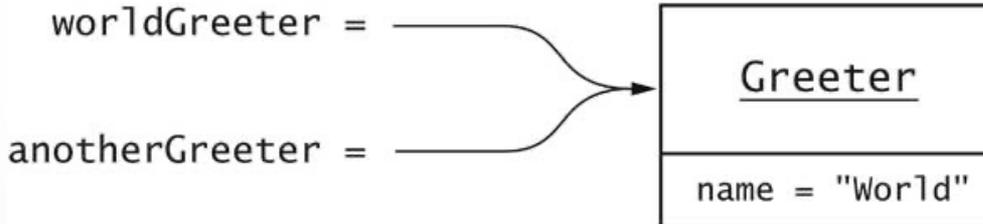
- An object value is always a reference to an object.

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



- Can have multiple references to the same object

```
Greeter anotherGreeter = worldGreeter;
```



- After applying mutator method, all references access modified object

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

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

The null References

- 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**

Parameter Passing

- Object reference on which you invoke a method is called **implicit parameters**.
- A method may have any number of **explicit parameters** that are supplied between parenthesis.

```
myGreeter.setName("Mars")
```

- Occasionally, you need to refer to the implicit parameter of a method by its special name (**this**).

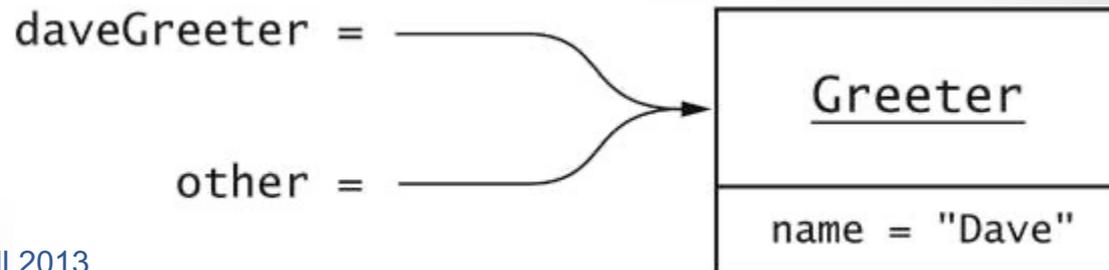
```
public void setName (String name)
{
    this.name = name;
}
```

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);
```



Parameter Passing

- 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;
```

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

```
Greeter daveGreeter = new Greeter("Dave");
```

```
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

- **Recommendation**

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

edu.sjsu.cs.cs151.alice.Greeter

java.util.ArrayList

javax.swing.JOptionPane

Packages

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

```
import java.util.Scanner;  
.  
.  
Scanner a; // i.e. java.util.Scanner
```

- Can import all classes from a package.

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

Packages

- Cannot import from multiple packages

```
import java.*.*; // NO
```

- If a class occurs in two imported packages, import is no help (such as class Date).

```
import java.util.*;  
import java.sql.*;
```

- You must use the full name:

```
java.util.Date date;
```

- 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 (NullPointerException)
- 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)
 - Handle or Catch the exception

Declaring Checked Exceptions

- 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.
- For debugging purposes you need to see the **stack trace**.

```
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.
- Strings positions start at 0.

```
String greeting = "Hello";
```

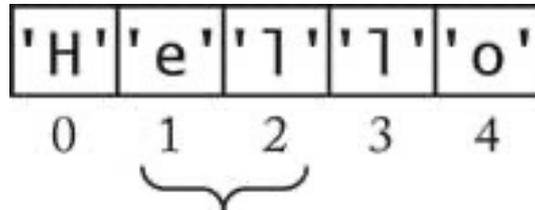
```
char ch = greeting.charAt(1); // sets ch to 'e'
```

- **length** method yields the number of characters in a String.
- "" is the empty string of length 0, different from **null**.
- "Hello".length() is 5.

Strings

- Substring method yields substrings:

`"Hello".substring(1, 3)` is `"el"`



- Since strings are objects, use **equals method** to compare strings

```
if (greeting.equals("Hello"))
```

- `==` only tests whether the object references are identical.

```
if ("Hello".substring(1, 3) == "el") ... // NO!
```

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

- Construct **Scanner** from input stream (e.g. System.in)

```
01: import java.util.Scanner;
02:
03: public class InputTester
04: {
05:     public static void main(String[] args)
06:     {
07:         Scanner in = new Scanner(System.in);
08:         System.out.print("How old are you?");
09:         int age = in.nextInt();
10:         age++;
11:         System.out.println("Next year, you'll be " + age);
12:     }
13: }
```

- If the user types input that is not a number, **InputMismatchException** is thrown.

hasNextInt, **hasNextDouble** test whether next token is a number.

Reading Input

- **next** reads next string (delimited by whitespace).
- **nextLine** reads next line.

```
Scanner in = new Scanner (new FileReader ("input.txt"));
While (in.hasNextLine( ) )
{
    String line = in.nextLine( );
    .....
}
```

The ArrayList<E> class

- **Generic class:** ArrayList<E> collects objects of type E.
- **E cannot be a primitive type.**

You can use an ArrayList<Date> **but NOT** ArrayList<int>

- **add** appends to the end of the array list.

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

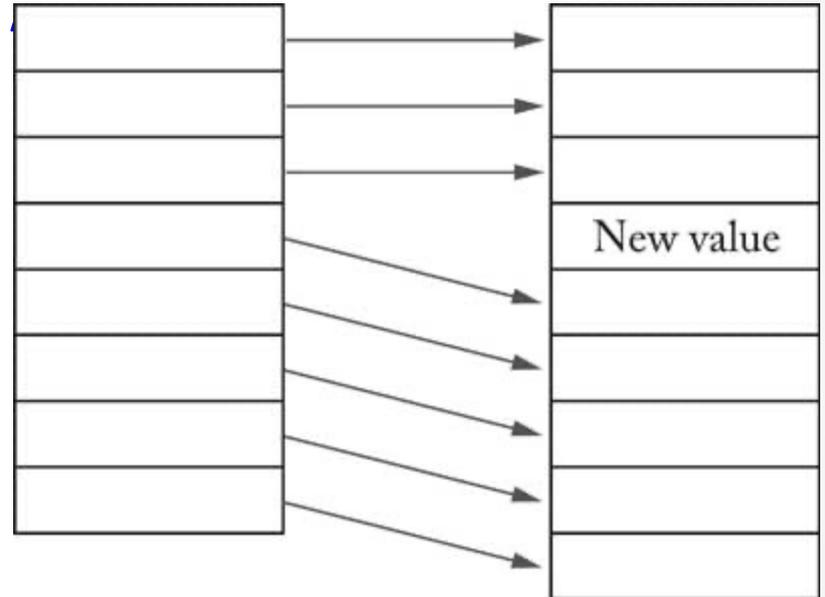
The ArrayList<E> class

- **get** gets an element; no need to cast to correct type:
`String country = countries.get(i);`
- **set** sets an element
`countries.set(1, "France");`
- **size** method yields number of elements
`for (int i = 0; i < countries.size(); i++) . .`
- Use "for each" loop
`for (String country : countries) . . .`
- Legal positions ranges from 0 to `size() - 1`.

The ArrayList<E> class

- Insert and remove elements in the middle

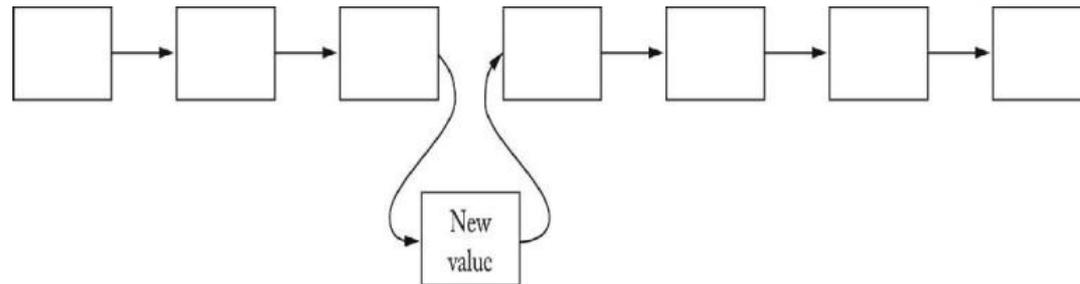
```
countries.add(1, "Germany");  
countries.remove(0);
```



- Not efficient--use linked lists if needed frequently

Linked Lists

- Efficient insertion and removal



- **add** appends to the end of the linked list.

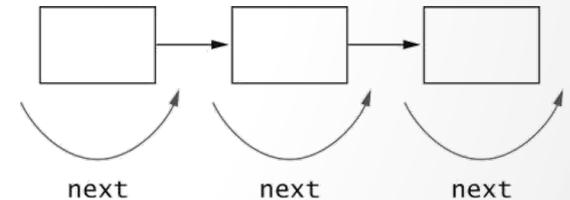
```
LinkedList<String> countries = new LinkedList<String>();  
countries.add("Belgium");  
countries.add("Italy");  
countries.add("Thailand");
```

- Use **iterators** to edit in the middle

List Iterators

- Iterator points between list elements.
- `next` retrieves element and advances iterator

```
ListIterator<String> iterator = countries.listIterator();  
while (iterator.hasNext())  
{  
    String country = iterator.next();  
    . . .  
}
```



- Or use "for each" loop:
`for (String country : countries)`

List Iterators

- **add** adds element before iterator position

```
iterator = countries.listIterator( );  
iterator.next( );  
iterator.add("France");
```

- **remove** removes element returned by last call to next

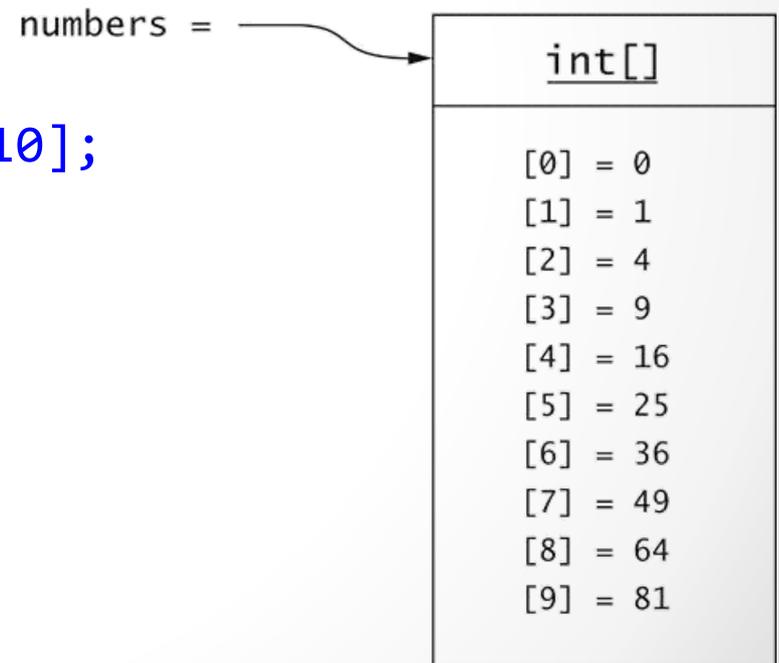
Example: Remove the second element of the countries list.

```
iterator = countries.listIterator( );  
iterator.next( );  
iterator.next( );  
iterator.remove;
```

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

- Array access with [] operator:

```
int n = numbers[ i ];
```

- **length** member yields number of elements

```
for (int i = 0; i < numbers.length; i++)
```

- Use "for each" loop

```
for (int n : numbers) . . .
```

Array

- Can have array of length 0; *not* the same as **null**:

```
numbers = new int[0];
```

- **Multidimensional array**

```
int[ ][ ] table = new int[10][20];  
int t = table[ row ] [ column ];
```

Static Fields

- Shared among all instances of a class.
- static field is more accurately called **class variable**.

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

- *factory method*

```
public static Greeter getRandomInstance()  
{  
    if (generator.nextBoolean())  
        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 (**camelCase**):

```
name  
sayHello
```

- **Classes:**

start with uppercase, use caps for new words (**PascalCase**):

```
Greeter  
ArrayList
```

- **Constants:**

use all caps, underscores to separate words

```
PI  
MAX_VALUES
```

Programming Style: Property Access

- Common to use **get/set** prefixes:

```
public 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 + "!";  
}
```

We will follow "Allman" style.

Programming Style: Fields

- Some programmers put fields before methods:

```
public class Greeter
{
    private String name;

    public Greeter(String aName) { . . . }
    . . .
}
```

- **All fields should be private.**
- Don't use default (package) visibility.

Programming Style: Miscellaneous

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

| Bad | Good |
|-------------------------------------|---------------------------------------|
| <code>if(x>Math.sqrt (y))</code> | <code>if (x > Math.sqrt(y))</code> |

- Don't use C-style arrays:

| Bad | Good |
|----------------------------|----------------------------|
| <code>int numbers[]</code> | <code>int[] numbers</code> |

- No magic numbers

| Bad | Good |
|-------------------------------------|--|
| <code>h = 31 * h + val[off];</code> | <code>final int HASH_MULTIPLIER = 31; h = HASH_MULTIPLIER * h + val[off];</code> |

End of Chapter 1