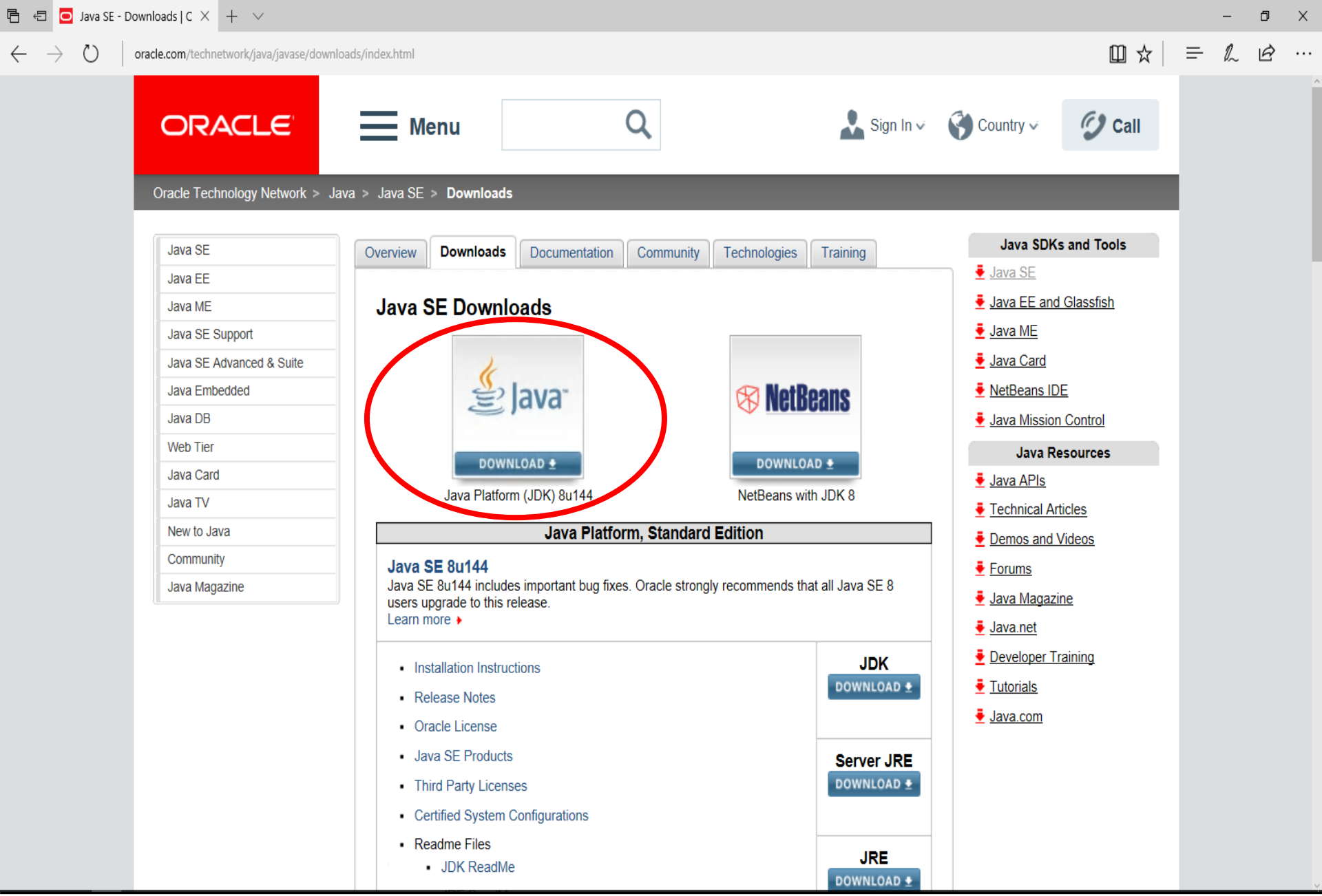
Java Tutorial

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

Java

- The Java **programming language** was created by Sun Microsystems, Inc. (Acquired by Oracle in January, 2010)
- It was introduced in 1995 and its popularity has grown quickly since
- A **programming language** specifies the words and symbols that we can use to write a program
- A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid program statements



http://www.oracle.com/technetwork/java/javase/downloads/index.html

Java Development Kit (JDK)

- javac - The Java Compiler
- java - The Java Interpreter

- javadoc - documentation generator
- javap - to print the Java bytecodes
- jdb - The Java Debugger
- appletviewer -Tool to run the applets
- …

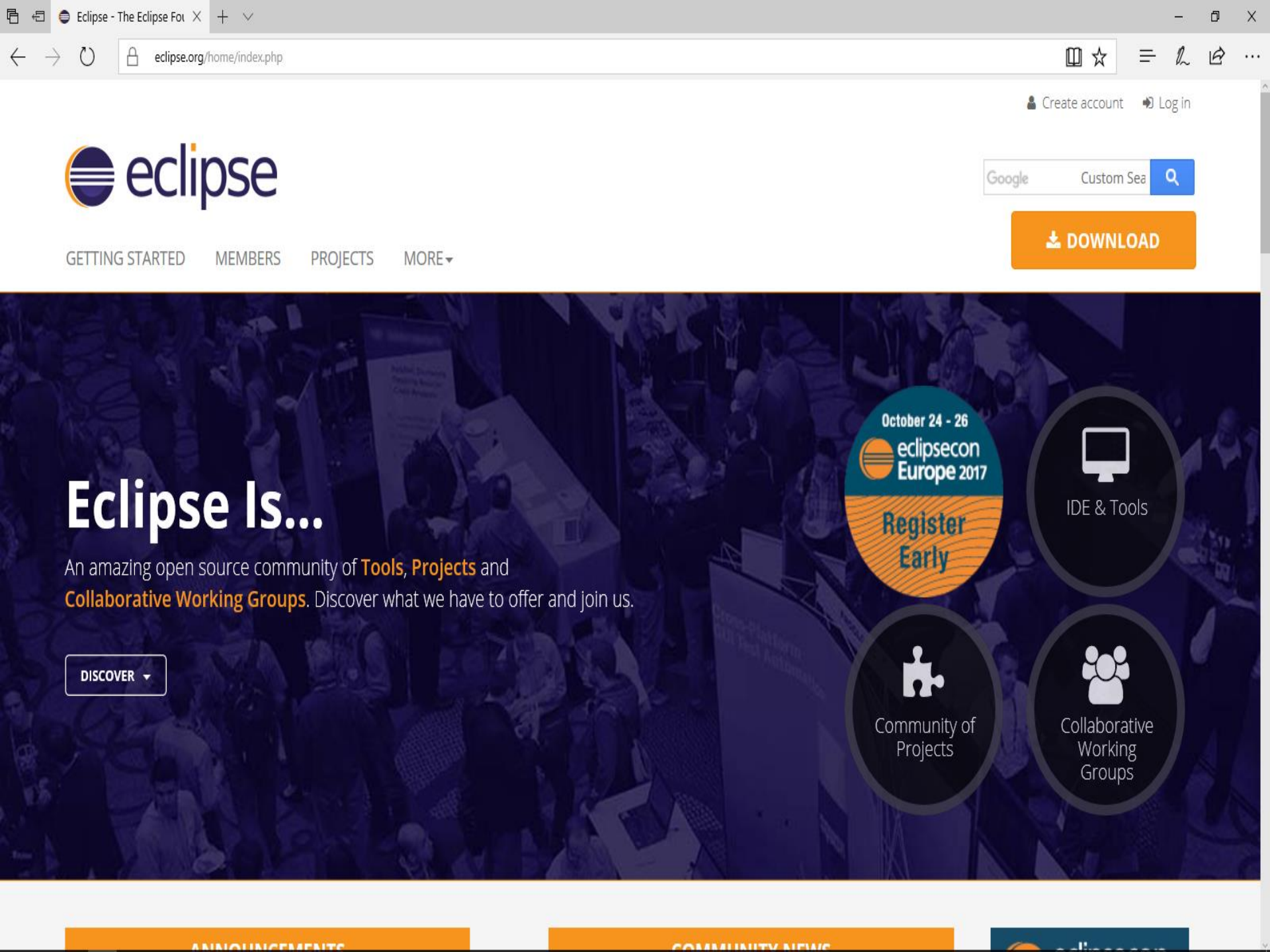
- http://docs.oracle.com/javase/8/docs/index.html

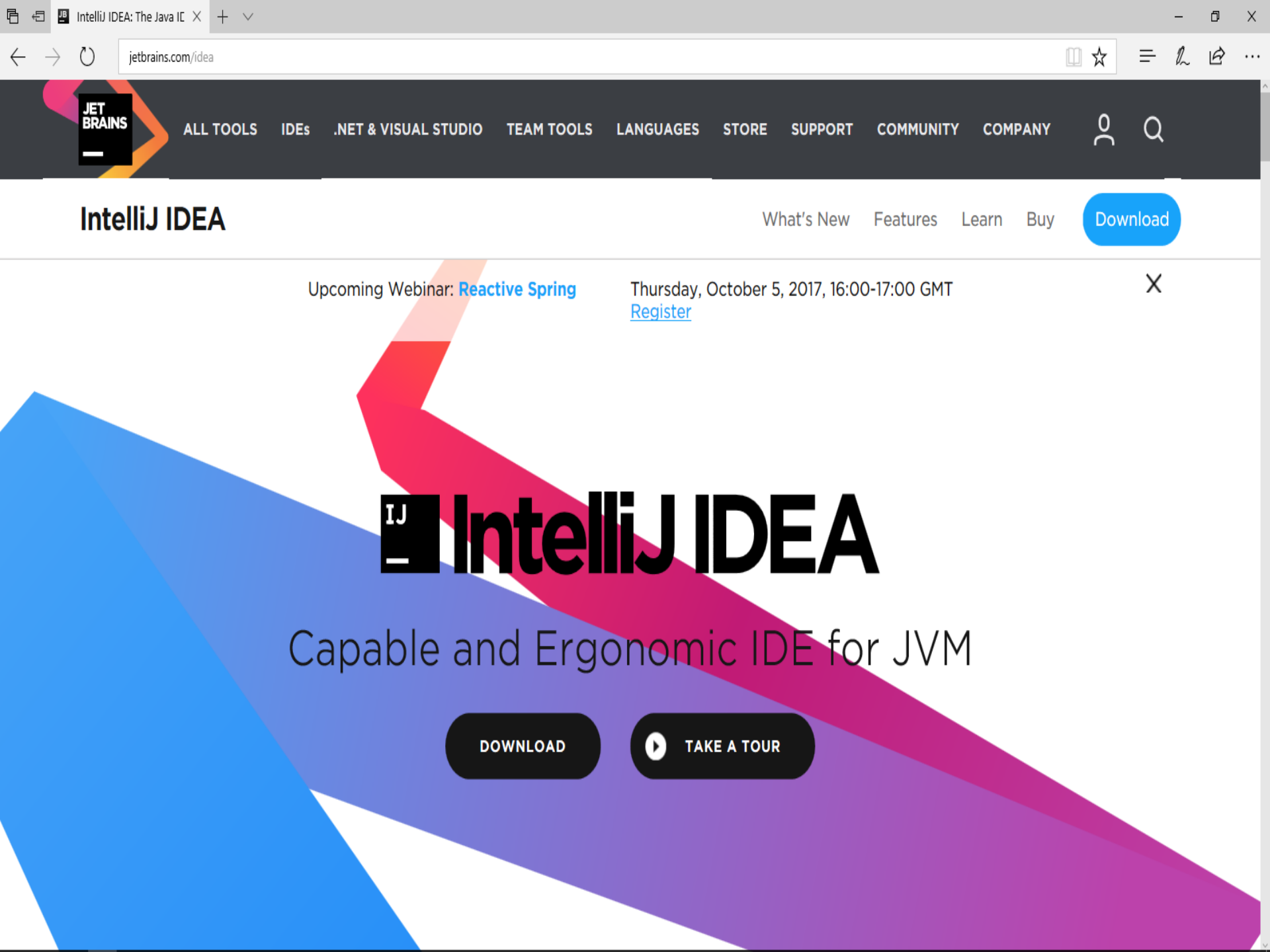
Development Environments

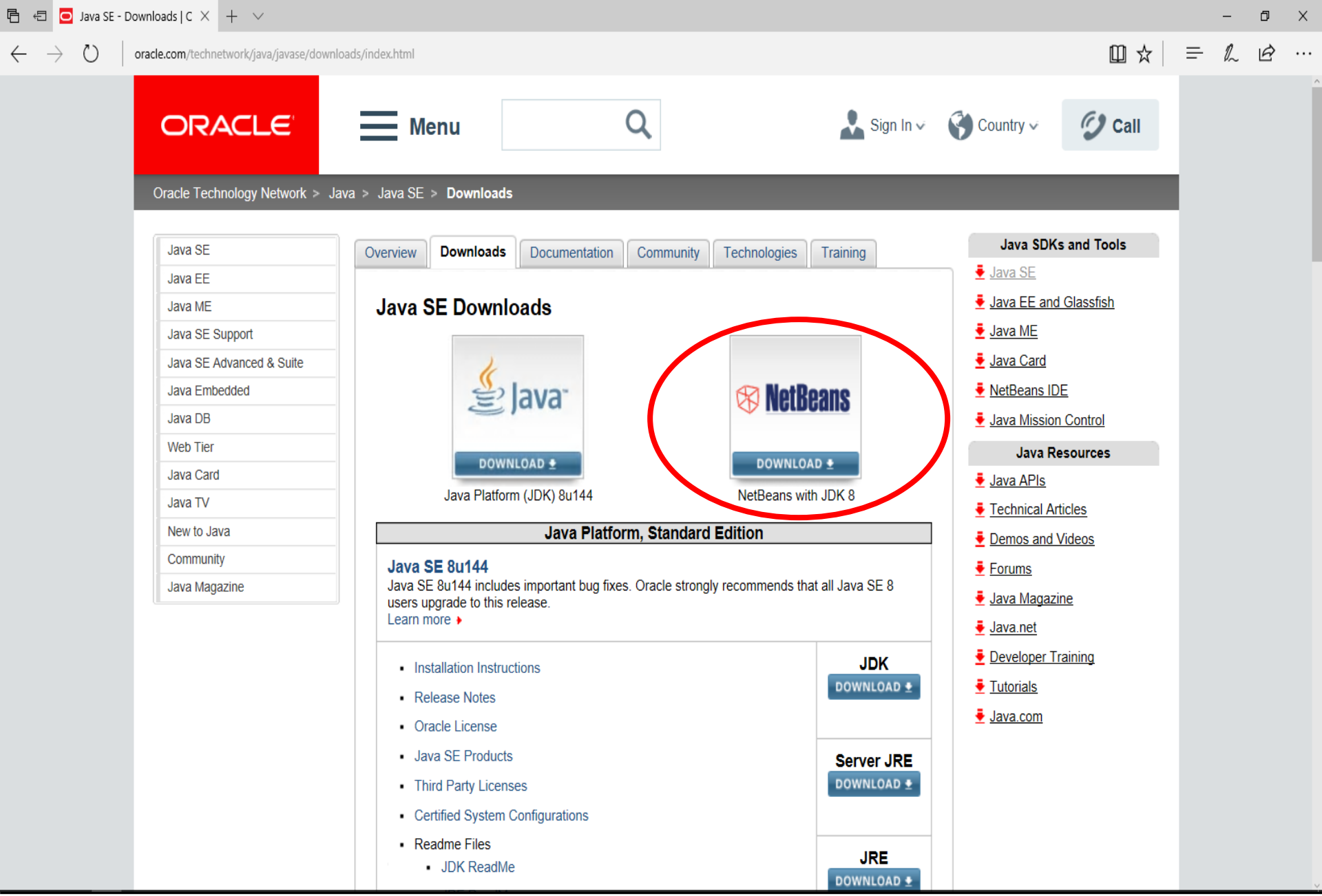
- There are many programs that support the development of Java software (see next slide), besides JDK & DrJava that you used in ECS 102, including:
  - **Eclipse**
  - IntelliJ
  - NetBeans
  - BlueJ
  - jGRASP
- Though the details of these environments differ, the **basic compilation and execution process** is essentially the same

Java has strong IDE support, due not only to its historical and economic importance, but also due to a combination of reflection and static-typing making it well-suited for IDE support. Some of the leading Java IDEs (such as IntelliJ and Eclipse) are also the basis for leading IDEs in other programming languages (e.g. for Python, IntelliJ is rebranded as PyCharm, and Eclipse has the PyDev plugin.)

IDE	License	Written in Java	Windows	Linux	macOS	Other platforms	GUI builder
Anjuta	GPL	No	No	Yes	No	FreeBSD	Yes
BlueJ	GPL2+GNU linking exception	Yes	Yes	Yes	Yes	Solaris	No
DrJava	Permissive	Yes	Yes	Yes	Yes	Solaris	No
Eclipse JDT IBM	EPL	Yes	Yes	Yes	Yes	FreeBSD, JVM, Solaris	Yes
Geany	GPL	No	Yes	Yes	Yes	FreeBSD, AIX, OpenBSD, Solaris, other Unix	No
Greenfoot	GPL	Yes	Yes	Yes	Yes	Solaris	No
IntelliJ IDEA	Community Edition: Apache License v2.0, Ultimate Edition: proprietary	Yes	Yes	Yes	Yes	FreeBSD, OpenBSD, Solaris	Yes
JBuilder	Proprietary	Yes	Yes	Yes	Yes	Solaris	Yes
JCreator	Proprietary	No	Yes	No	No		No
JDeveloper	Proprietary (freeware)	Yes	Yes	Yes	Yes	generic JVM	Yes
jGRASP	Proprietary (freeware)	Yes	Yes	Yes	Yes		No
KDevelop	GPL	No	No	Yes	No	Solaris	Yes
MyEclipse	Proprietary	Yes	Yes	Yes	Yes	FreeBSD, JVM, Solaris	Yes
NetBeans	CDDL, GPL2	Yes	Yes	Yes	Yes	OpenBSD, Solaris	Yes
Rational Application Developer	Proprietary	Yes	Yes	Yes	No	AIX, Solaris	Yes
Servoy	Proprietary	Yes	Yes	Yes	Yes	Solaris	Yes
SlickEdit	Proprietary	No	Yes	Yes	Yes	Solaris, Solaris SPARC, AIX, HP-UX	No
Understand	Proprietary	No	Yes	Yes	Yes	Solaris	Yes
Xcode (Apple)	Proprietary	No	No	No	Yes		Yes


Eclipse - The Eclipse Fou
eclipse.org/home/index.php
Create account
Log in
eclipse
Google
Custom Sea
GETTING STARTED
MEMBERS
PROJECTS
MORE
DOWNLOAD
Eclipse Is...
An amazing open source community of Tools, Projects and
Collaborative Working Groups. Discover what we have to offer and join us.
DISCOVER
October 24 - 26
eclipsecon
Europe 2017
Register
Early
IDE & Tools
Community of
Projects
Collaborative
Working
Groups


IntelliJ IDEA: The Java IC
jetbrains.com/idea
JET BRAINS
ALL TOOLS
IDEs
.NET & VISUAL STUDIO
TEAM TOOLS
LANGUAGES
STORE
SUPPORT
COMMUNITY
COMPANY
IntelliJ IDEA
What's New
Features
Learn
Buy
Download
Upcoming Webinar: Reactive Spring
Thursday, October 5, 2017, 16:00-17:00 GMT
Register
IJ
IntelliJ IDEA
Capable and Ergonomic IDE for JVM
DOWNLOAD
TAKE A TOUR



http://www.oracle.com/technetwork/java/javase/downloads/index.html

Choose page language

NetBeans | NetBeans IDE | NetBeans Platform | Enterprise | Plugins | Docs & Support | Community | Search

NetBeans IDE

Fits the Pieces Together

Quickly and easily develop desktop, mobile and web applications with Java, JavaScript, HTML5, PHP, C/C++ and more.

NetBeans IDE is FREE, open source, and has a worldwide community of users and developers.

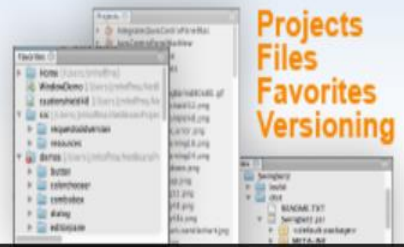
NetBeans IDE 8.1

Learn More | Download

Featured News: Code Sharing for NetBeans Developers

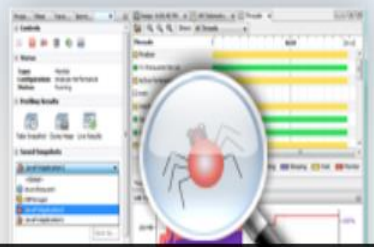
See All News

Easy & Efficient Project Management



Rapid User Interface Development

Write Bug Free Code



BlueJ

A free Java Development Environment designed for beginners, used by millions worldwide. Find out more...

"One of my favourite IDEs out there is BlueJ"
— James Gosling, creator of Java.

Created by University of Kent

Supported by ORACLE

Download and Install

Version 3.1.7, released 23 February 2016 (what's new)

Windows

BlueJ Installer with Java 8 JDK (recommended)

OR

1. Oracle JDK
2. BlueJ Installer

You can also download BlueJ as a standalone bundle (zip) which you can run from a USB stick.

jGRASP

An Integrated Development Environment with Visualizations for Improving Software Comprehensibility

Home
Download
Contact Us
Team Members
Resources
Archive
Privacy Policy
Support jGRASP

Documentation

jGRASP Help
On-line Papers
Tips
FAQ
Known Bugs
Version History
Future Plans
Accessibility
License

Intro Videos

Getting Started
Canvas (new)
Interactions

Tutorials (PDF)

Overview (2.0)
Installation
Getting Started (2.0)

Current jGRASP release is version 2.0.3 (August 16, 2016).

If you haven't used the viewer canvas for Java, you will find this video useful: viewer canvas.

Download jGRASP

Institutions Using jGRASP

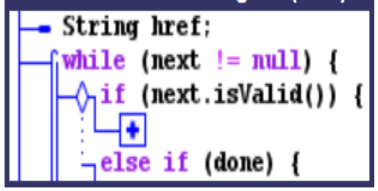
Support jGRASP

About jGRASP

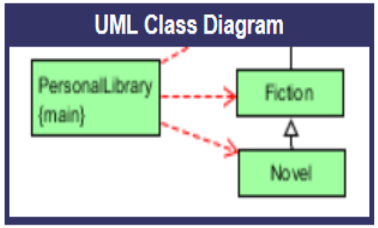
jGRASP is a lightweight development environment, created specifically to provide automatic generation of software visualizations to improve the comprehensibility of software. jGRASP is implemented in Java, and runs on all platforms with a Java Virtual Machine (Java version 1.5 or higher). jGRASP produces Control Structure Diagrams (CSDs) for Java, C, C++, Objective-C, Python, Ada, and VHDL; Complexity Profile Graphs (CPGs) for Java and Ada; UML class diagrams for Java; and has dynamic object viewers and a viewer canvas that work in conjunction with an integrated debugger and workbench for Java. The viewers include a data structure identifier mechanism which recognizes objects that represent traditional data structures such as stacks, queues, linked lists, binary trees, and hash tables, and then displays them in an intuitive textbook-like presentation view.

jGRASP is developed by the Department of Computer Science and Software Engineering in the Samuel Ginn College of Engineering at Auburn University.

Control Structure Diagram (CSD)



UML Class Diagram



New Releases

Version 2.0.3 supports pinch-zoom and Ctrl (or Cmd) scroll wheel zoom.

Multiple editing window tab panes (or virtual desktops, if you use them that way) are available in version 2.0.3.

Java Program Structure

- In the Java programming language:
  - A program is made up of one or more **classes**
  - A class contains one or more **methods**
  - A method contains program **statements**

- A Java application always contains a method called **main**

Java Program Structure

- A Java program consists of, at least, a **definition** of a class.
  - The name of the class (ClassName) and the name of the file (ClassName.java) must be identical.
- The class definition has a **header** (public class ClassName) followed by a **body** inside braces.
- In Java, every application must contain a main method whose **signature** is:

  public static void main(String[] args)

header

```
public class HelloWorldApp {
        public static void main(String[] args)
        {
            System.out.println("Hello World!");  // Display the string.
         }
}
```

body

Access Levels

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Comments

- Comments should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take three forms:

```
// this comment runs to the end of the line

/*  this comment runs to the terminating
    symbol, even across line breaks        */

/** this is a javadoc comment   */
```

Output

```
A quote by Abraham Lincoln:
Whatever you are, be a good one.
```

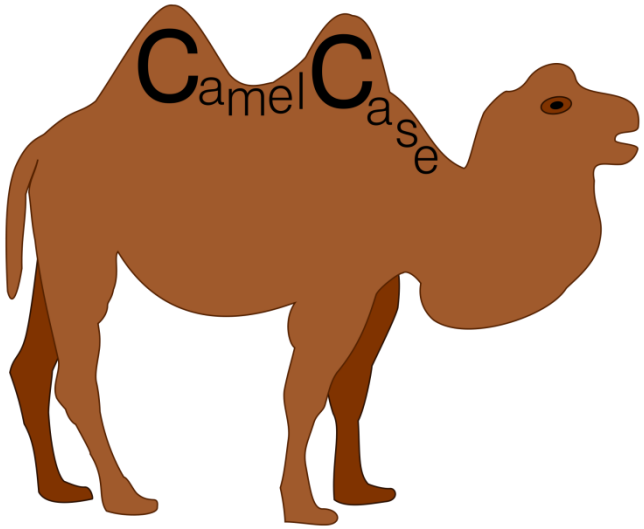
```
//********                                                    **********
//  Lincol
//
//  Demons
//*************************************************************************

public class Lincoln
{
   //-----------------------------------------------------------------
   //  Prints a presidential quote.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      System.out.println("A quote by Abraham Lincoln:");

      System.out.println("Whatever you are, be a good one.");
   }
}
```

Identifiers

- **Identifiers** are the "words" in a program
- A Java identifier can be made up of letters, digits, the underscore character (_), and the dollar sign ($)
- Identifiers cannot begin with a digit
- Java is case sensitive: Total, total, and TOTAL are different identifiers
- By convention, programmers use different case styles for different types of identifiers, such as
  - **Upper case** for class names - Lincoln
  - **All upper case** for constants - MAXIMUM

CamelCase

- **CamelCase** (**camel case**) is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter.
  - Camel case may start with a capital or, especially in programming languages, with a lowercase letter.
  - Common examples are PowerPoint or iPhone.



http://en.wikipedia.org/wiki/CamelCase

Identifiers

- Sometimes the programmer chooses the identifier (such as Lincoln)
- Sometimes we are using another programmer's code, so we use the identifiers that he or she chose (such as **println**)
- Often we use special identifiers called **reserved words** that already have a predefined meaning in the language
  - A reserved word cannot be used in any other way

Reserved Words

- The Java reserved words:

`abstract`	`else`	`interface`	`switch`
`assert`	`enum`	`long`	`synchronized`
`boolean`	`extends`	`native`	`this`
`break`	`false`	`new`	`throw`
`byte`	`final`	`null`	`throws`
`case`	`finally`	`package`	`transient`
`catch`	`float`	`private`	`true`
`char`	`for`	`protected`	`try`
`class`	`goto`	`public`	`void`
`const`	`if`	`return`	`volatile`
`continue`	`implements`	`short`	`while`
`default`	`import`	`static`	
`do`	`instanceof`	`strictfp`	
`double`	`int`	`super`	

Quiz

Which of the following are valid Java identifiers?

- grade
- quizGrade
- NetworkConnection
- frame2
- 3rdTestScore
- MAXIMUM
- MIN_CAPACITY
- student#
- Shelves1&2

Quiz

Which of the following are valid Java identifiers?

grade	Valid
quizGrade	Valid
NetworkConnection	Valid
frame2	Valid
3rdTestScore	Invalid – cannot begin with a digit
MAXIMUM	Valid
MIN_CAPACITY	Valid
student#	Invalid – cannot contain the '#' character
Shelves1&2	Invalid – cannot contain the '&' character

White Space

- Spaces, blank lines, and tabs are called **white space**
- White space is used to separate words and symbols in a program
- Extra white space is ignored
- A valid Java program can be formatted many ways
- Programs should be formatted to enhance **readability**, using consistent indentation
- See Lincoln2.java and Lincoln3.java

Lincoln2.java

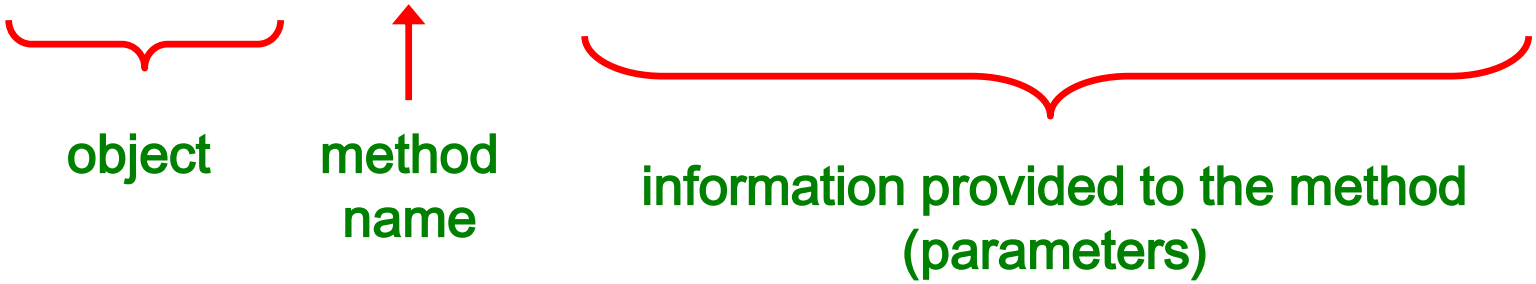
```
//********************************************************************
//  Lincoln2.java       Author: Lewis/Loftus
//
//  Demonstrates a poorly formatted, though valid, program.
//********************************************************************

public class Lincoln2{public static void main(String[]args){
System.out.println("A quote by Abraham Lincoln:");
System.out.println("Whatever you are, be a good one.");}}
```

```
//********************************************************************
//  Lincoln3.java       Author: Lewis/Loftus
//
//  Demonstrates another valid program that is poorly formatted.
//********************************************************************

        public      class
    Lincoln3
   {
              public
   static
       void
  main
       (
String
          []
    args                    )
  {
  System.out.println       (
"A quote by Abraham Lincoln:"         )
  ;       System.out.println
          (
      "Whatever you are, be a good one."
     )
  ;
}
        }
```

The println Method

- In the Lincoln program, we invoked the **println** method to print a character string
- The **System.out** object represents a destination (the monitor screen) to which we can send output

```
System.out.println ("Whatever you are, be a good one.");
```



compact1, compact2, compact3

java.lang

Class System

java.lang.Object
    java.lang.System

```
public final class System
extends Object
```

The `System` class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the `System` class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Since:

`JDK1.0`

Field Summary

Fields

Modifier and Type	Field and Description
`static PrintStream`	`err` The "standard" error output stream.
`static InputStream`	`in` The "standard" input stream.
`static PrintStream`	`out` The "standard" output stream.

The print Method

- The **System.out** object provides another service as well
  - The **print** method is similar to the println method, except that it does not advance to the next line
  - Therefore anything printed after a print statement will appear on the same line
  - See Countdown.java (next slide)

println vs. println

Output

```
Three... Two... One... Zero... Liftoff!
Houston, we have a problem.
```

```
//****
//  Co
//
//  De
//****

public class Countdown
{
   //-----------------------------------------------------------------
   //  Prints two lines of output representing a rocket countdown.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.print("Three... ");
      System.out.print("Two... ");
      System.out.print("One... ");
      System.out.print("Zero... ");
      System.out.println("Liftoff!");  // appears on first output line
      System.out.println("Houston, we have a problem.");
   }
}
```

String Concatenation

- The string concatenation operator (+) is used to append one string to the end of another

  "Peanut butter " + "and jelly"

- It can also be used to append a number to a string
- A string literal cannot be broken across two lines in a program
- See Facts.java (next slide)

```
//********************************************************************
//  Facts.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the string concatenation operator and the
//  automatic conversion of an integer to a string.
//********************************************************************

public class Facts
{
   //-----------------------------------------------------------------
   //  Prints various facts.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      // Strings can be concatenated into one long string
      System.out.println("We present the following facts for your "
                         + "extracurricular edification:");

      System.out.println();

      // A string can contain numeric digits
      System.out.println("Letters in the Hawaiian alphabet: 12");
```

continue

continue

```
        // A numeric value can be concatenated to a string
        System.out.println("Dialing code for Antarctica: " + 672);

        System.out.println("Year in which Leonardo da Vinci invented "
                           + "the parachute: " + 1515);

        System.out.println("Speed of ketchup: " + 40 + " km per year");
    }
}
```

Output

```
We present the following facts for your extracurricular edification:

Letters in the Hawaiian alphabet: 12
Dialing code for Antarctica: 672
Year in which Leonardo da Vinci invented the parachute: 1515
Speed of ketchup: 40 km per year
```

contin

```
        System.out.println("Year in which Leonardo da Vinci invented "
                           + "the parachute: " + 1515);

        System.out.println("Speed of ketchup: " + 40 + " km per year");
    }
}
```

String Concatenation

- The + operator is also used for arithmetic addition
- The function that it performs depends on the type of the information on which it operates
- If both operands are strings, or if one is a string and one is a number, it performs string concatenation
- If both operands are numeric, it adds them
- The + operator is evaluated left to right, but parentheses can be used to force the order
- See Addition.java

```
//********************************************************************
//  Addition.java       Author: Lewis/Loftus
//
//  Demonstrates the difference between the addition and string
//  concatenation operators.
//********************************************************************

public class Addition
{
   //-----------------------------------------------------------------
   //  Concatenates and adds two numbers and prints the results.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.println("24 and 45 concatenated: " + 24 + 45);

      System.out.println("24 and 45 added: " + (24 + 45));
   }
}
```

```
//**********                                                  **********
//  Addition.
//
//  Demonstra                                                 dition and string
//  concatena
//*************************************************************************

public class Addition
{
   //-----------------------------------------------------------------
   //  Concatenates and adds two numbers and prints the results.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.println("24 and 45 concatenated: " + 24 + 45);

      System.out.println("24 and 45 added: " + (24 + 45));
   }
}
```

Output

```
24 and 45 concatenated: 2445
24 and 45 added: 69
```

Quiz

What output is produced by the following?

```
System.out.println("X: " + 25);
System.out.println("Y: " + (15 + 50));
System.out.println("Z: " + 300 + 50);
```

Quiz

What output is produced by the following?

```
System.out.println("X: " + 25);
System.out.println("Y: " + (15 + 50));
System.out.println("Z: " + 300 + 50);
```

```
X: 25
Y: 65
Z: 30050
```

Escape Sequences

- What if we wanted to print the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

```
System.out.println("I said "Hello" to you.");
```

- An **escape sequence** is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\)

```
System.out.println("I said \"Hello\" to you.");
```

Escape Sequences

- Some Java escape sequences:

Escape Sequence	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

- See Roses.java (next slide)

Escape Sequences

```
//********************************************************************
//  Roses.java       Author: Lewis/Loftus
//
//  Demonstrates the use of escape sequences.
//********************************************************************

public class Roses
{
   //-----------------------------------------------------------------
   //  Prints a poem (of sorts) on multiple lines.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.println("Roses are red,\n\tViolets are blue,\n" +
         "Sugar is sweet,\n\tBut I have \"commitment issues\",\n\t" +
         "So I'd rather just be friends\n\tAt this point in our " +
         "relationship.");
   }
}
```

Escape Sequences

```
//****
//  Ro
//
//  De
//****

public
{
   //-
   //
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      System.out.println ("Roses are red,\n\tViolets are blue,\n" +
         "Sugar is sweet,\n\tBut I have \"commitment issues\",\n\t" +
         "So I'd rather just be friends\n\tAt this point in our " +
         "relationship.");
   }
}
```

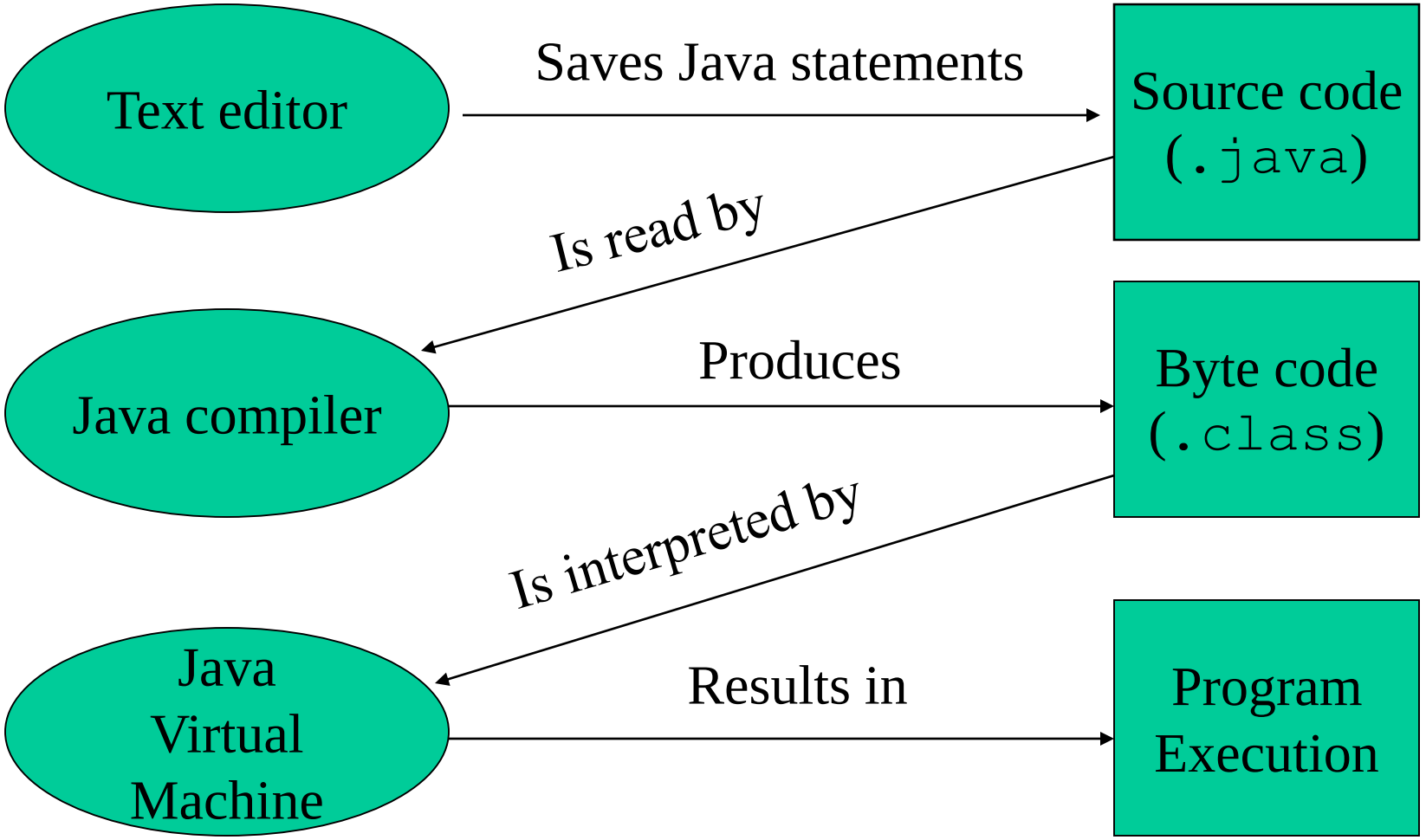
Output

```
Roses are red,
        Violets are blue,
Sugar is sweet,
        But I have "commitment issues",
        So I'd rather just be friends
        At this point in our relationship.
```

The Java Virtual Machine

- Most compilers translate source code into **executable** files containing **machine code**.
- The Java compiler translates a Java source file into a file that contains **byte code** instructions.
- Byte code instructions are the machine language of the **Java Virtual Machine (JVM)** and cannot be directly executed directly by the CPU.
- Byte code files end with the *.class* file extension.
- The JVM executes instructions as they are read.
- JVM is often called an **interpreter**, and hence often referred to as an **interpreted language**.

Program Development Process



Command-line Arguments

- The main method accepts a single argument: an array of elements of type String. The following line is called the **signature** of the main method:

  public static void main(String[] args)

- This array is the mechanism through which the runtime system passes information to your application. For example:

  java *MyApp arg1 arg2*

- Each string in the array is called a **command-line argument**.

- Command-line arguments let users affect the operation of the application without recompiling it.

- The HelloWorld! program ignores its command-line arguments, but you should be aware of the fact that such arguments do exist.

Questions and Exercises: Getting Started

Questions

Question 1: When you compile a program written in the Java programming language, the compiler converts the human-readable source file into platform-independent code that a Java Virtual Machine can understand. What is this platform-independent code called?

Question 2: Which of the following is *not* a valid comment:

a. `/** comment */`

b. `/* comment */`

c. `/* comment`

d. `// comment`

Question 3: What is the first thing you should check if you see the following error at runtime:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
HelloWorldApp.java.
```

Question 4: What is the correct signature of the `main` method?

Question 5: When declaring the `main` method, which modifier must come first, `public` or `static`?

Question 6: What parameters does the `main` method define?

Exercises

Exercise 1: Change the `HelloWorldApp.java` program so that it displays `Hola Mundo!` instead of `Hello World!`.

Exercise 2: You can find a slightly modified version of `HelloWorldApp` here: `HelloWorldApp2.java`

ORACLE Java Documentation

The Java™ Tutorials

Download Ebooks
Download JDK
Search Java Tutorials

The Java Tutorials are practical guides for programmers who want to use the Java programming language to create applications. They include hundreds of complete, working examples, and dozens of lessons. Groups of related lessons are organized into "trails".

The Java Tutorials primarily describe features in Java SE 8. For best results, download JDK 8.

What's New

The Java Tutorials are continuously updated to keep up with changes to the Java Platform and to incorporate feedback from our readers.

This release of the tutorial corresponds to the JDK 8u101 release.

Two new processing limit properties, `entityReplacementLimit` and `maxXMLNameLimit`, have been added to JAXP. See Processing Limits for more information.

Trails Covering the Basics

These trails are available in book form as *The Java Tutorial, Sixth Edition*. To buy this book, refer to the box to the right.

- Getting Started — An introduction to Java technology and lessons on installing Java development software and using it to create a simple program.
- Learning the Java Language — Lessons describing the essential concepts and features of the Java Programming Language.
- Essential Java Classes — Lessons on exceptions, basic input/output, concurrency, regular expressions, and the platform environment.
- Collections — Lessons on using and extending the Java Collections Framework.
- Date-Time APIs — How to use the `java.time` pages to write date and time code.
- Deployment — How to package applications and applets using JAR files, and deploy them using Java Web Start and Java Plug-in.
- Preparation for Java Programming Language Certification — List of available training and tutorial resources.

Creating Graphical User Interfaces

Not sure where to start?
See Learning Paths

Tutorial Contents

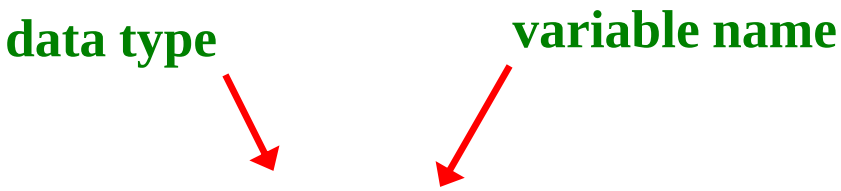
Really Big Index

Tutorial Resources

- View the Java Tutorials Online (Last Updated 7/19/2016).
- The Java Tutorials' Blog has news and updates about the Java SE tutorials.
- Download the latest Java Tutorials bundle.

Variables

- A **variable** is a name for a location in memory that holds a value
- A **variable declaration** specifies the variable's name and the type of information that it will hold




```
int total;
int count, temp, result;
```

Multiple variables can be created in one declaration

Identifiers (Revisited)

- Variable names are identifiers.
  - Identifiers are made up of
    - upper case letters
    - lower case letters
    - digit 0-9
    - _
    - $
    - First symbol can't be a digit

Variable Names

- Variable names should be descriptive.
  - Descriptive names allow the code to be more readable; therefore, the code is more maintainable.
- Which of the following is more descriptive?

```
double tr = 0.0725;
double salesTaxRate = 0.0725;
```

- Java programs should be **self-documenting**.

Variable Initialization

- A variable can be given an initial value in the declaration

```
int sum = 0;
int base = 32, max = 149;
```

- When a variable is referenced in a program, its current value is used
- See PianoKeys.java (next slide)

Variables

```
//********************************************************************
//  PianoKeys.java       Author: Lewis/Loftus
//
//  Demonstrates the declaration, initialization, and use of an
//  integer variable.
//********************************************************************

public class PianoKeys
{
   //-----------------------------------------------------------------
   //  Prints the number of keys on a piano.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int keys = 88;
      System.out.println("A piano has " + keys + " keys.");
   }
}
```

Variables

Output

```
A piano has 88 keys.
```

```
//********************************************************************
//  PianoKeys.java
//
//  Demonstrates the declaration, initialization, and use of an
//  integer variable.
//********************************************************************

public class PianoKeys
{
   //-----------------------------------------------------------------
   //  Prints the number of keys on a piano.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int keys = 88;
      System.out.println("A piano has " + keys + " keys.");
   }
}
```

Assignment

- An **assignment statement** changes the value of a variable
- The assignment operator is the = sign

```
value = 5;
```

- The value that was in total is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type

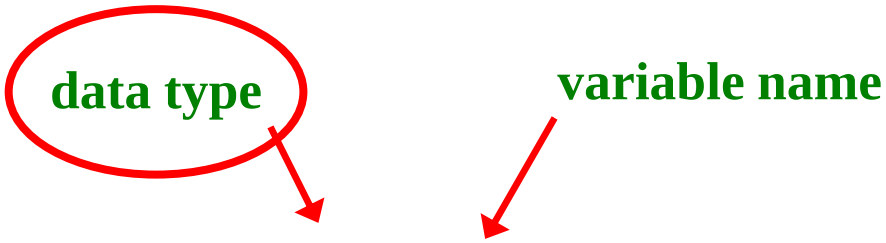
Constants

- A **constant** is an identifier that is similar to a variable except that it holds the same value during its entire existence
- As the name implies, it is constant, not variable
- The compiler will issue an error if you try to change the value of a constant
- In Java, we use the final modifier to declare a constant

final int MIN_HEIGHT = 69;

Variables

- A **variable** is a name for a location in memory that holds a value
- A **variable declaration** specifies the variable's name and the type of information that it will hold




```
int total;
int count, temp, result;
```

Multiple variables can be created in one declaration

Primitive Data Types

- There are eight primitive data types in Java
  - Four of them represent integers:
    - byte, short, int, long
  - Two of them represent floating point numbers:
    - float, double
  - One of them represents characters:
    - char
  - And one of them represents boolean values:
    - boolean

Numeric Primitive Data Types

- The difference between the numeric primitive types is their size and the values they can store:

Type	Storage	Min Value	Max Value
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
float	32 bits	$+/- 3.4 \times 10^{38}$ with 7 significant digits	
double	64 bits	$+/- 1.7 \times 10^{308}$ with 15 significant digits	

Characters

- A char variable stores a single character
- Character literals are delimited by single quotes:

'a' 'X' '7' '$' ',' '\n'

- Example declarations:

```
char topGrade = 'A';
char terminator = ';', separator = ' ';
```

- Note the difference between a primitive character variable, which holds only one character, and a String object, which can hold multiple characters

Characters

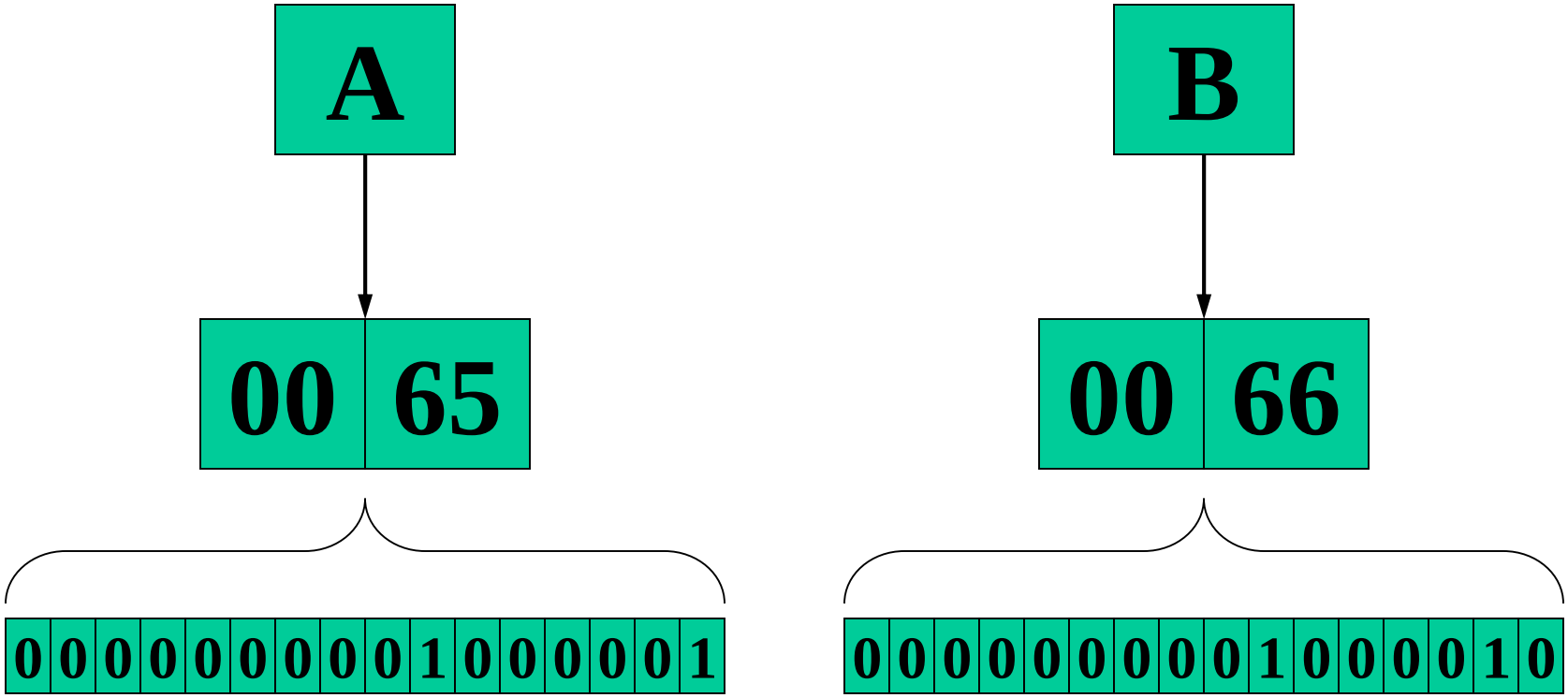
- The **ASCII character set** is older and smaller than Unicode, but is still quite popular
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters	**A, B, C, …**
lowercase letters	**a, b, c, …**
punctuation	**period, semi-colon, …**
digits	**0, 1, 2, …**
special symbols	**&, |, \, …**
control characters	**carriage return, tab, ...**

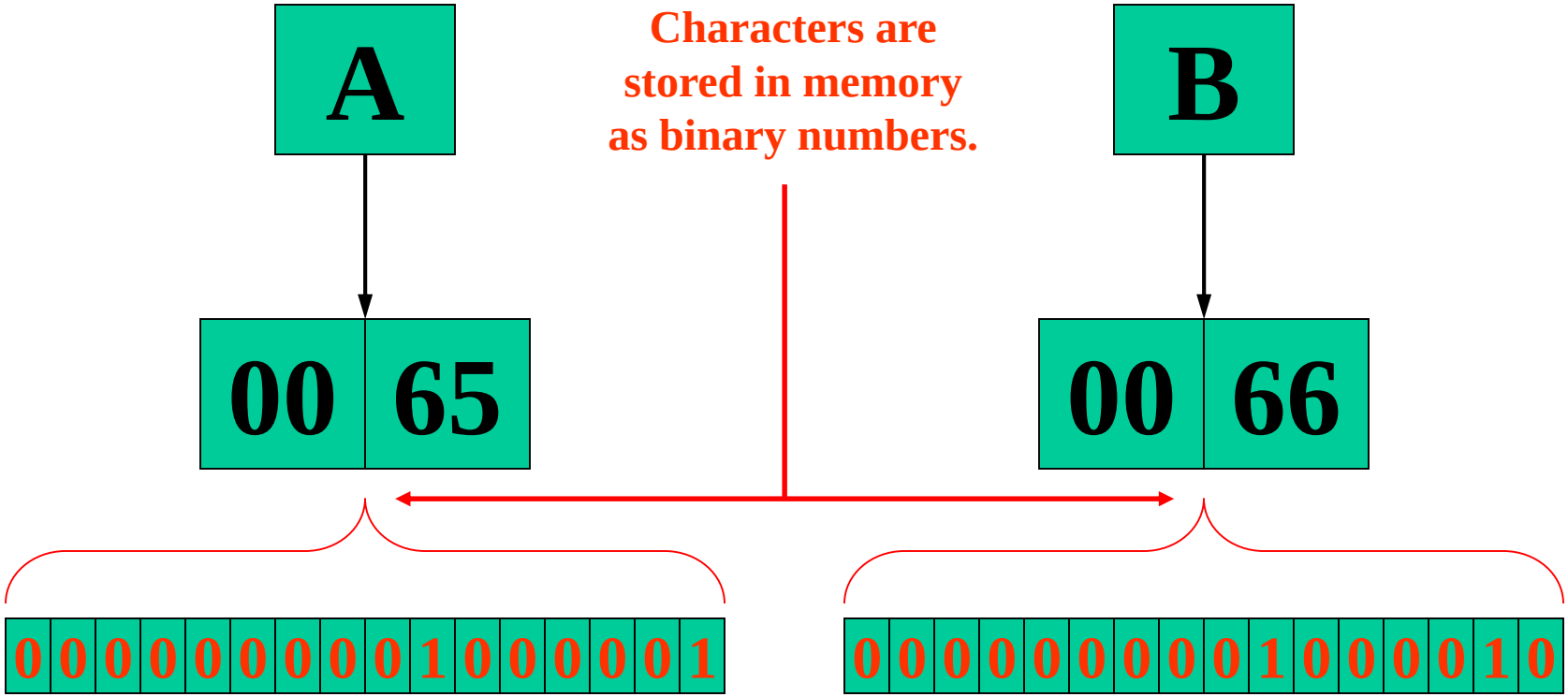
Character Sets

- A **character set** is an ordered list of characters, with each character corresponding to a unique number
- A char variable in Java can store any character from the **Unicode** character set
- The Unicode character set uses sixteen bits per character, allowing for 65,536 unique characters
- It is an international character set, containing symbols and characters from many world languages

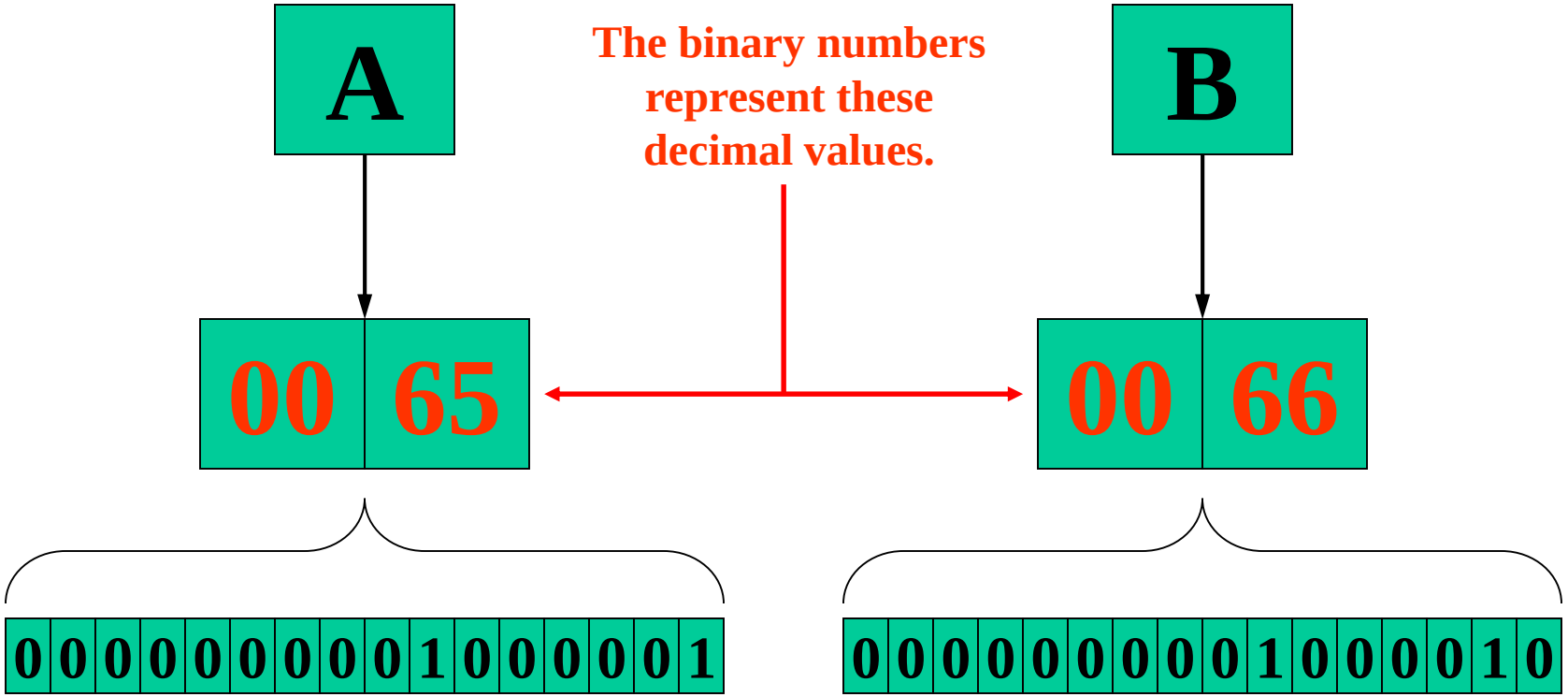
Characters: Unicode



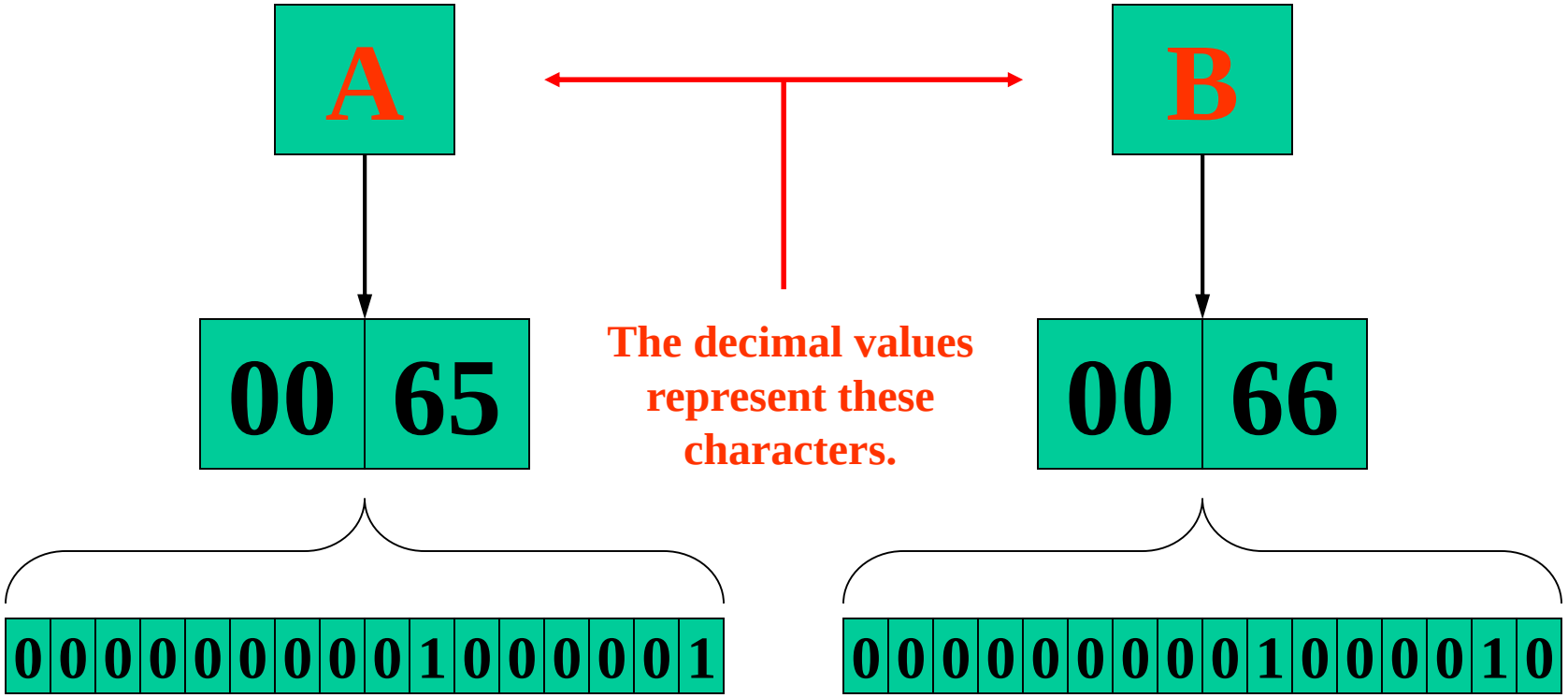
Characters: Unicode



Characters: Unicode



Characters: Unicode



Boolean

- A **boolean** value represents a true or false condition
- The reserved words **true** and **false** are the only valid values for a boolean type

```
boolean done = false;
```

- A boolean variable can also be used to represent any two states, such as a light bulb being on or off
- Although the **boolean** data type represents 1 bit of information, its ‘size’ is not precisely defined in Java. (Usually it’s either 1 byte like **byte** or 4 bytes like **int**.)

Expressions

- An **expression** is a combination of one or more **operators** and **operands**
- **Arithmetic expressions** compute numeric results and make use of the arithmetic operators:

Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Remainder (modulo)	**%**

- If either or both operands are floating point values, then the result is a floating point value

Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

14 / 3 equals 4

8 / 12 equals 0

- The remainder operator (%) returns the remainder after dividing the first operand by the second

14 % 3 equals 2

8 % 12 equals 8

Quick Check

What are the results of the following expressions?

12 / 2

12.0 / 2.0

10 / 4

10 / 4.0

4 / 10

4.0 / 10

12 % 3

10 % 3

3 % 10

Quick Check

What are the results of the following expressions?

12 / 2 = 6

12.0 / 2.0 = 6.0

10 / 4 = 2

10 / 4.0 = 2.5

4 / 10 = 0

4.0 / 10 = 0.4

12 % 3 = 0

10 % 3 = 1

3 % 10 = 3

Operator Precedence

- Operators can be combined into larger expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication (*), division(/), and remainder (%) are evaluated before addition (+), subtraction (-), and string concatenation (+)
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order

Quick Check

In what order are the operators evaluated in the following expressions?

```
a + b + c + d + e
```

```
a + b * c - d / e
```

```
a / (b + c) - d % e
```

```
a / (b * (c + (d - e)))
```

Quick Check

In what order are the operators evaluated in the following expressions?

```
a + b + c + d + e
  1   2   3   4
```

```
a + b * c - d / e
  3   1   4   2
```

```
a / (b + c) - d % e
  2    1    4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Operator Precedence

Operator Precedence	
Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? : Ex: minVal = (a < b) ? a : b;
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

```
answer  =  sum / 4 + MAX * lowest;
```

4 1 3 2

Then the result is stored in the variable on the left hand side

Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable

First, one is added to the original value of count

```
count  =  count + 1;
```

Then the result is stored back into count (overwriting the original value)

Increment and Decrement

- The increment (++) and decrement (--) operators use only one operand
- The statement

```
count++;
```

is functionally equivalent to

```
count = count + 1;
```

Increment and Decrement

- The increment and decrement operators can be applied in **postfix form**:

count++

- or **prefix form**:

++count

- When used as part of a larger expression, the two forms can have different effects

- Because of their subtleties, the increment and decrement operators should be used with care

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Java provides **assignment operators** to simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```

Assignment Operators

- There are many assignment operators in Java, including the following:

Operator	Example	Equivalent	Value of variable after operation
+=	`x += 5;`	`x = x + 5;`	The old value of **x** plus 5.
-=	`y -= 2;`	`y = y - 2;`	The old value of **y** minus 2
***=**	`z *= 10;`	`z = z * 10;`	The old value of **z** times 10
/=	`a /= b;`	`a = a / b;`	The old value of **a** divided by **b**.
%=	`c %= 3;`	`c = c % 3;`	The remainder of the division of the old value of **c** divided by 3.

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
  - The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total - MIN) % num;
```

is equivalent to

```
result = result / ((total - MIN) % num);
```

Data Conversion

- Sometimes it is convenient to convert data from one type to another
  - For example, in a particular situation we may want to treat an *integer* as a *floating point value*
  - These conversions do not change the type of a variable or the value that's stored in it – they only convert a value as part of a computation

Data Conversion

- **Widening conversions** are safest because they tend to go from a small data type to a larger one (such as a short to an int)

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

- **Narrowing conversions** can lose information because they tend to go from a large data type to a smaller one (such as an int to a short)

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

- In Java, data conversions can occur in three ways:
  - assignment conversion
  - promotion
  - casting

Assignment Conversion

- **Assignment conversion** occurs when a value of one type is assigned to a variable of another
  - Example:

```
int dollars = 20;
double money = dollars;
```

- Only widening conversions can happen via assignment
- Note that the value or type of **dollars** did not change

Promotion

- **Promotion** happens automatically when operators in expressions convert their operands
  - Example:

```
int count = 12;
double sum = 490.27;
result = sum / count;
```

- The value of **count** is converted to a floating point value to perform the division calculation
- Concatenation of a string and a number is also considered as a kind of promotion:

```
System.out.println("A piano has " + keys + " keys.");
```

Casting

- **Casting** is the most powerful, and dangerous, technique for conversion
  - Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted

```
int total = 50;
float result = (float) total / 6;
```

- Without the cast, the fractional part of the answer would be lost

The Scanner Class

- Programs generally need input on which to operate – interactive programming
- The **Scanner** class provides convenient methods for reading input values of various types
  - A Scanner object can be set up to read input from various sources, including the user typing values on the keyboard
  - Keyboard input is represented by the **System.in** object

Object-Oriented Programming

- Java is an object-oriented programming language
  - As the term implies, an **object** is a fundamental entity in a Java program
  - **Objects** can be used effectively to represent real-world entities
    - For instance, an object might represent a particular employee in a company
    - Each employee object handles the processing and data management related to that employee

compact1, compact2, compact3

java.lang

Class Object

java.lang.Object

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

`JDK1.0`

See Also:

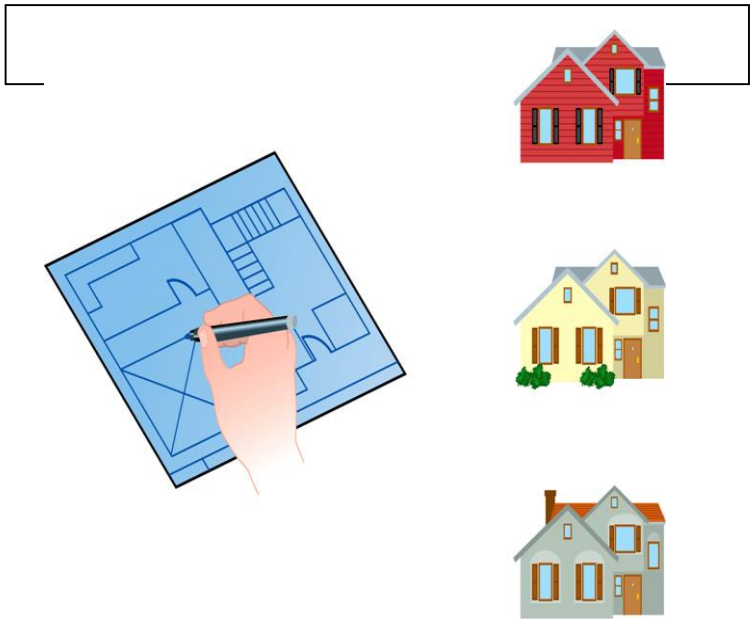
`Class`

Constructor Summary

Constructors

Classes

- An object is defined by a **class**
- A **class** is the blueprint of an object
  - A class represents a concept, and an object represents the embodiment of that concept
  - The class uses methods to define the behaviors of the object
  - The class that contains the main method of a Java program represents the entire program
  - Multiple objects can be created from the same class



Objects and Classes

A class
(the concept)

An object
(the embodiment)

John's Bank Account
Balance: $5,257

Bill's Bank Account
Balance: $1,245,069

Mary's Bank Account
Balance: $16,833

Multiple objects
from the same class

Back to Scanner

- The following line creates a Scanner object that reads from the keyboard:

```
Scanner scan = new Scanner(System.in);
```

- Once created, the Scanner object can be used to invoke various input methods, such as:

```
answer = scan.nextLine();
```

- The Scanner class is part of the **java.util** class library, and must be imported into a program to be used

- The **nextLine** method reads all of the input until the end of the line is found

- See Echo.java (next slide)

```
//********************************************************************
//  Echo.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the nextLine method of the Scanner class
//  to read a string from the user.
//********************************************************************

import java.util.Scanner;

public class Echo
{
   //-----------------------------------------------------------------
   //  Reads a character string from the user and prints it.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String message;
      Scanner scan = new Scanner(System.in);

      System.out.println("Enter a line of text:");

      message = scan.nextLine();

      System.out.println("You entered: \"" + message + "\"");
   }
}
```

The Scanner class is part of the java.util class library, and must be imported into a program to be used

The import Declaration

- When you want to use a class from a **package** (see next slide), you could use its fully qualified name

```
java.util.Scanner
```

- Or you can **import** the class, and then use just the class name

```
import java.util.Scanner;
```

- To import all classes in a particular package, you can use the * wildcard character

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

Packages

- For purposes of accessing them, classes in the Java are organized into **packages**
- Examples:

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

The import Declaration

- All classes of the **java.lang** package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the **System** class explicitly in earlier programs.
- The Scanner class, on the other hand, is part of the **java.util** package, and therefore must be imported

Sample Run

```
Enter a line of text:
You want fries with that?
You entered: "You want fries with that?"
```

```
//*********
//  Echo.ja
//
//  Demonst
//  to read
//*********

import java.util.Scanner;

public class Echo
{
   //-----------------------------------------------------------------
   //  Reads a character string from the user and prints it.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String message;
      Scanner scan = new Scanner(System.in);

      System.out.println("Enter a line of text:");

      message = scan.nextLine();

      System.out.println("You entered: \"" + message + "\"");
   }
}
```

Input Tokens

- Unless specified otherwise, **whitespace** is used to separate the elements (called **tokens**) of the input
  - White space includes space characters, tabs, new line characters
- There is a **next** method in the Scanner class that reads the next input token and returns it as a string
- Methods such as **nextInt** and **nextDouble** read data of particular types
- See GasMileage.java (next slide)

```
//********************************************************************
//  GasMileage.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the Scanner class to read numeric data.
//********************************************************************

import java.util.Scanner;

public class GasMileage
{
   //-----------------------------------------------------------------
   //  Calculates fuel efficiency based on values entered by the
   //  user.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int miles;
      double gallons, mpg;

      Scanner scan = new Scanner(System.in);
```

continue

continue

```
        System.out.print("Enter the number of miles: ");
        miles = scan.nextInt();

        System.out.print("Enter the gallons of fuel used: ");
        gallons = scan.nextDouble();

        mpg = miles / gallons;

        System.out.println("Miles Per Gallon: " + mpg);
    }
}
```

Sample Run

```
Enter the number of miles: 328
Enter the gallons of fuel used: 11.2
Miles Per Gallon: 29.28571428571429
```

continue

```
        System.out.print("Enter the number of miles: ");
        miles = scan.nextInt();

        System.out.print("Enter the gallons of fuel used: ");
        gallons = scan.nextDouble();

        mpg = miles / gallons;

        System.out.println("Miles Per Gallon: " + mpg);
    }
}
```

Table 2-17 Some of the `Scanner` class methods

Method	Example and Description

`nextByte`	**Example Usage:**
	`byte x;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a byte value: ");`
	`x = keyboard.nextByte();`
	Description: Returns input as a byte.

`nextDouble`	**Example Usage:**
	`double number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a double value: ");`
	`number = keyboard.nextDouble();`
	Description: Returns input as a double.

`nextFloat`	**Example Usage:**
	`float number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a float value: ");`
	`number = keyboard.nextFloat();`

`nextInt`

Example Usage:

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

Description: Returns input as an `int`.

`nextLine`

Example Usage:

```
String name;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your name: ");
name = keyboard.nextLine();
```

Description: Returns input as a `String`.

`nextLong`

Example Usage:

```
long number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a long value: ");
number = keyboard.nextLong();
```

Description: Returns input as a `long`.

`nextShort`

Example Usage:

```
short number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a short value: ");
number = keyboard.nextShort();
```

Description: Returns input as a `short`.

The String Class

- Because strings are so common, we don't have to use the new operator to create a String object:

```
String value = "Hello";
```

  - This special syntax works only for strings
  - A variable can also be created using the **new** keyword:

```
String value = new String("Hello");
```

- Each string literal (enclosed in double quotes) represents a String object

The String Methods

- Since String is a class, objects that are instances of it have methods.
- One of those methods is the **length** method:

```
stringSize = value.length();
```

  - This statement runs the length method on the object pointed to by the value variable.
- See example: StringLength.java

StringLength.java

```
// This program demonstrates the String class's length method.

public class StringLength
{
   public static void main(String[] args)
   {
      String name = "Herman";
      int stringSize;

      stringSize = name.length();
      System.out.println(name + " has " + stringSize + " characters.");
   }
}
```

String Methods

- String objects are **immutable**, meaning that they cannot be changed.
- Fortunately, the String class contains many methods that can manipulate the original version of the String object, and help you create new versions of it.
- See example: StringMethods.java

StringMethods.java

```
// This program demonstrates a few of the String methods.

public class StringMethods {
   public static void main(String[] args)    {
      String message = "Java is Great Fun!";
      String upper = message.toUpperCase();
      String lower = message.toLowerCase();
      char letter = message.charAt(2);
      int stringSize = message.length();

      System.out.println(message);
      System.out.println(upper);
      System.out.println(lower);
      System.out.println(letter);
      System.out.println(stringSize);
   }
}
```

String Indexes

- It is occasionally helpful to refer to a particular character within a string
  - This can be done by specifying the character's numeric **index**
  - The indexes begin at zero in each string
  - In the string "Hello", the character 'H' is at index 0 and the 'o' is at index 4

Table 2-15 A few `String` class methods

Method	Description and Example

`charAt(index)`	The argument *index* is an `int` value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type `char`.
	Example:
	`char letter;`
	`String name = "Herman";`
	`letter = name.charAt(3);`
	After this code executes, the variable `letter` will hold the character 'm'.

`length()`	This method returns the number of characters in the string. The return value is of the type `int`.
	Example:
	`int stringSize;`
	`String name = "Herman";`
	`stringSize = name.length();`
	After this code executes, the `stringSize` variable will hold the value 6.

`toLowerCase()`	This method returns a new string that is the lowercase equivalent of the string contained in the calling object.
	Example:
	`String bigName = "HERMAN";`
	`String littleName = bigName.toLowerCase();`
	After this code executes, the object referenced by `littleName` will hold the string "herman".

`toUpperCase()`	This method returns a new string that is the uppercase equivalent of the string contained in the calling object.
	Example:
	`String littleName = "herman";`
	`String bigName = littleName.toUpperCase();`
	After this code executes, the object referenced by `bigName` will hold the string "HERMAN".

`String(String str)`
Constructor: creates a new string object with the same characters as `str`.

`char charAt(int index)`
Returns the character at the specified `index`.

`int compareTo(String str)`
Returns an integer indicating if this string is lexically before (a negative return value), equal to (a zero return value), or lexically after (a positive return value), the string `str`.

`String concat(String str)`
Returns a new string consisting of this string concatenated with `str`.

`boolean equals(String str)`
Returns true if this string contains the same characters as `str` (including case) and false otherwise.

`boolean equalsIgnoreCase(String str)`
Returns true if this string contains the same characters as `str` (without regard to case) and false otherwise.

`int length()`
Returns the number of characters in this string.

`String replace(char oldChar, char newChar)`
Returns a new string that is identical with this string except that every occurrence of `oldChar` is replaced by `newChar`.

`String substring(int offset, int endIndex)`
Returns a new string that is a subset of this string starting at index `offset` and extending through `endIndex-1`.

`String toLowerCase()`
Returns a new string identical to this string except all uppercase letters are converted to their lowercase equivalent.

`String toUpperCase()`
Returns a new string identical to this string except all lowercase letters are converted to their uppercase equivalent.

FIGURE 3.1 Some methods of the `String` class

Primitive vs. Reference Variables

- Primitive variables actually contain the value that they have been assigned.

```
int number = 25;
```

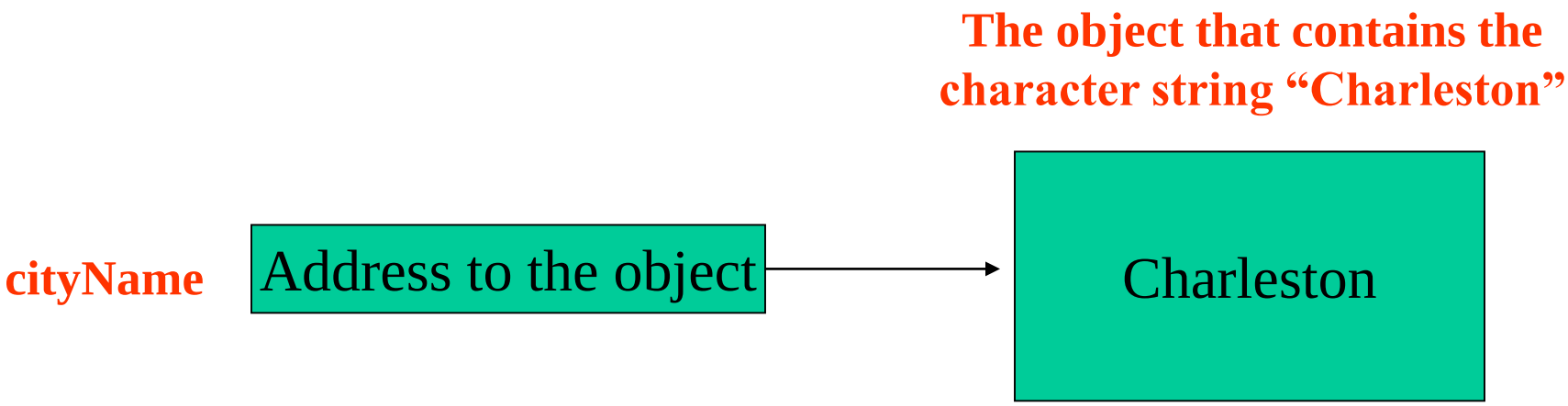
  - The value 25 will be stored in the memory location associated with the variable number.

- Objects are not stored in variables, however. Objects are **referenced** by variables.

Primitive vs. Reference Variables

- When a variable references an object, it contains the memory address of the object's location.
- Then it is said that the variable **references** the object.

```
String cityName = "Charleston";
```



The Math Class

- The methods of the Math class are **static methods** (also called **class methods**)

- Static methods are invoked through the class name – no object of the Math class is needed

```
value = Math.cos(90) + Math.sqrt(delta);
```

Unlike an **instance method**:

```
Scanner scan = new Scanner(System.in);

answer = scan.nextLine();
```

```
static int abs(int num)
```
Returns the absolute value of num.

```
static double acos(double num)
static double asin(double num)
static double atan(double num)
```
Returns the arc cosine, arc sine, or arc tangent of num.

```
static double cos(double angle)
static double sin(double angle)
static double tan(double angle)
```
Returns the angle cosine, sine, or tangent of angle, which is measured in radians.

```
static double ceil(double num)
```
Returns the ceiling of num, which is the smallest whole number greater than or equal to num.

```
static double exp(double power)
```
Returns the value e raised to the specified power.

```
static double floor(double num)
```
Returns the floor of num, which is the largest whole number less than or equal to num.

```
static double pow(double num, double power)
```
Returns the value num raised to the specified power.

```
static double random()
```
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

```
static double sqrt(double num)
```
Returns the square root of num, which must be positive.

```
static int abs(int num)
```
Returns the absolute value of num.

```
static double acos(double num)
static double asin(double num)
static double atan(double num)
```
Returns the arc cosine, arc sine, or arc tangent of num.

```
static double cos(double angle)
static double sin(double angle)
static double tan(double angle)
```
Returns the angle cosine, sine, or tangent of angle, which is measured in radians.

```
static double ceil(double num)
```
Returns the ceiling of num, which is the smallest whole number greater than or equal to num.

```
static double exp(double power)
```
Returns the value e raised to the specified power.

```
static double floor(double num)
```
Returns the floor of num, which is the largest whole number less than or equal to num.

```
static double pow(double num, double power)
```
Returns the value num raised to the specified power.

```
static double random()
```
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

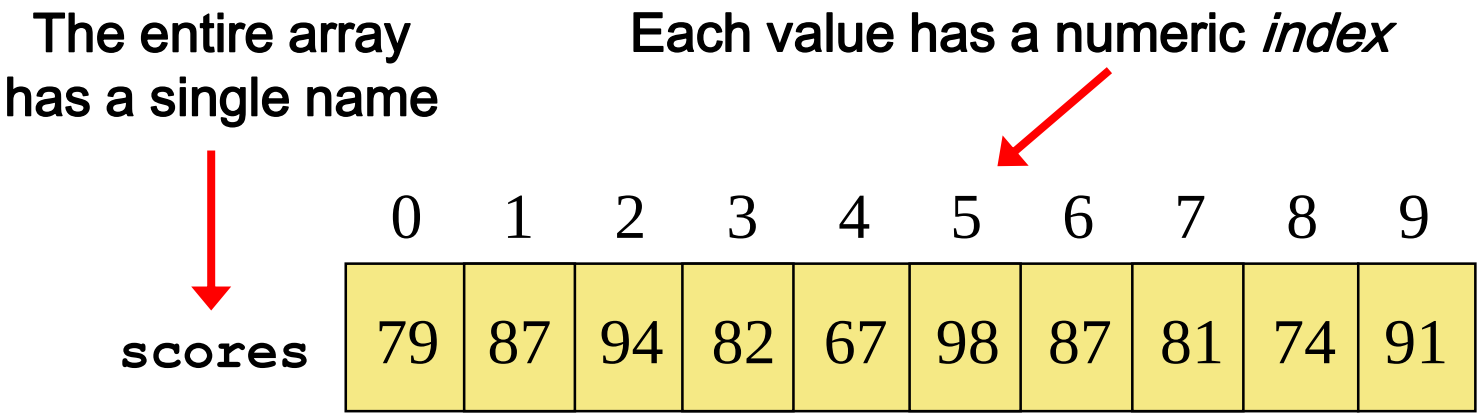
```
static double sqrt(double num)
```
Returns the square root of num, which must be positive.

Control Flow Statements

- Decision-making statements
  - if-then
  - if-then-else
  - switch
- Looping statements
  - for
  - while
  - do-while
- Branching statements
  - break
  - continue
  - return

Arrays

- An array is an ordered list of values:



An array of size N is indexed from zero to N-1

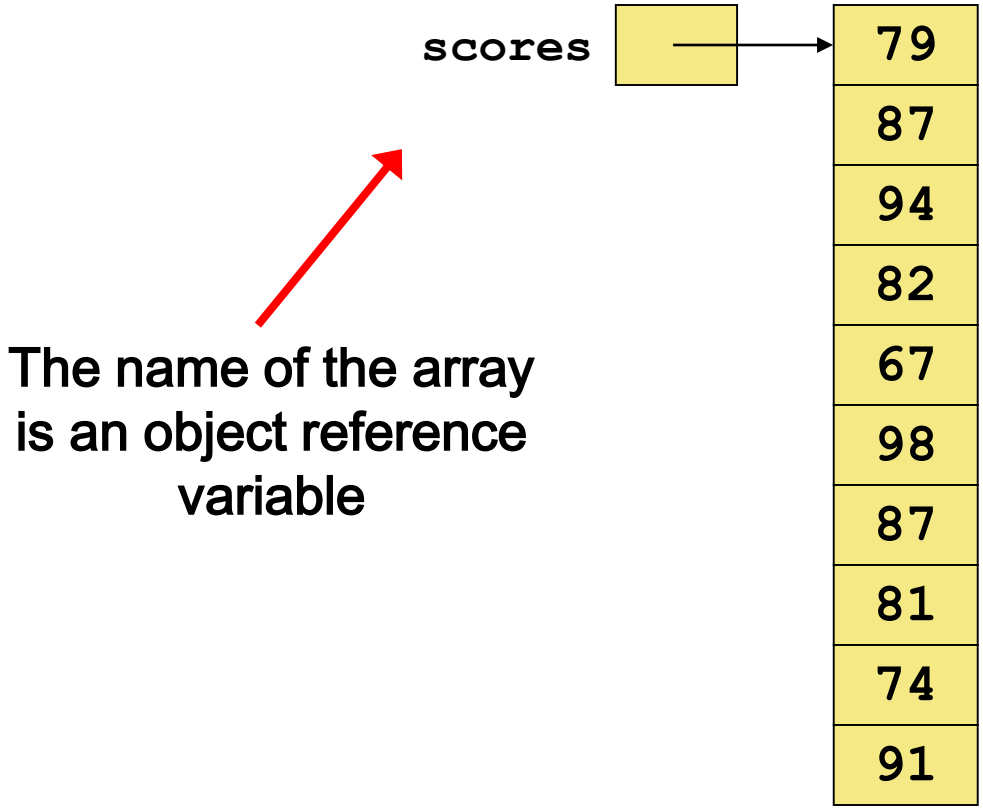
This array holds 10 values that are indexed from 0 to 9

Arrays

- The values held in an array are called **array elements**
- An array stores multiple values of the same type – the **element type**
  - The element type can be a primitive type or an object reference
  - Therefore, we can create an array of integers, an array of characters, an array of **String**s or other objects, etc.

Arrays

- In Java, the array itself is an object that must be instantiated



Declaring Arrays

- The scores array could be declared as follows:

int[] scores = new int[10];

  - The type of the variable scores is int[] (an array of integers)
  - Note that the array type does not specify its size, but each object of that type has a specific size
  - The reference variable scores is set to a new array object that can hold 10 integers

Using Arrays

- The for loop can be used when processing array elements:

```
for (int index = 0; index < 10; index++)
        System.out.println(scores[index])
```

- The **for-each** version of the for loop can be used when processing array elements: (aka. **Enhanced For Loop**)

```
for (int score : scores)
   System.out.println(score);
```

  - This is only appropriate when processing all array elements starting at index 0. It can't be used to set the array values

Bounds Checking

- Once an array is created, it has a fixed size. An index used in an array reference must specify a valid element
  - That is, the index value must be in range 0 to N-1
  - The Java interpreter throws an **ArrayIndexOutOfBoundsException** if an array index is out of bounds
  - This is called automatic **bounds checking**
  - It's common to introduce **off-by-one errors** when using arrays:

problem

```
for (int index=0; index <= 10; index++)
   codes[index] = index*50 + epsilon;
```

Bounds Checking

- Each array object has a public constant called length that stores the size of the array
  - It is referenced using the array name:

scores.length

  - Note that length holds the number of elements, not the largest index

Alternate Array Syntax

- The brackets of the array type can be associated with the element type or with the name of the array
- Therefore the following two declarations are equivalent:

```
double[] prices;
double prices[];
```

- The first format generally is more readable and should be used

Initializer Lists

- An **initializer list** can be used to instantiate and fill an array in one step
  - The values are delimited by braces and separated by commas

```
int[] scores = {147, 323, 89, 933, 540, 269, 97, 114, 298, 476};

char[] grades = {'A', 'B', 'C', 'D', 'F'};
```

  - Note that when an initializer list is used:
    - the **new** operator is not used
    - no size value is specified - the size of the array is determined by the number of items in the list
  - An initializer list can be used only in the array declaration

Creating 2-Dimensional Arrays

- To declare a two-dimensional array for this sales data, you simply list two sets of empty brackets, like this:

  double sales[][];

  Here, sales is a two-dimensional array of type double.

- To actually create the array, you use the new keyword and provide lengths for each set of brackets:

  sales = new double[5][4];

  - The first dimension specifies that the sales array has 5 elements - 5 rows in the table.
  - The second dimension specifies that each of those elements has an array of type double with 4 elements - 4 columns in the table.

- Combined:

  double[][] sales = new double[5][4];

Writing Your Own Classes

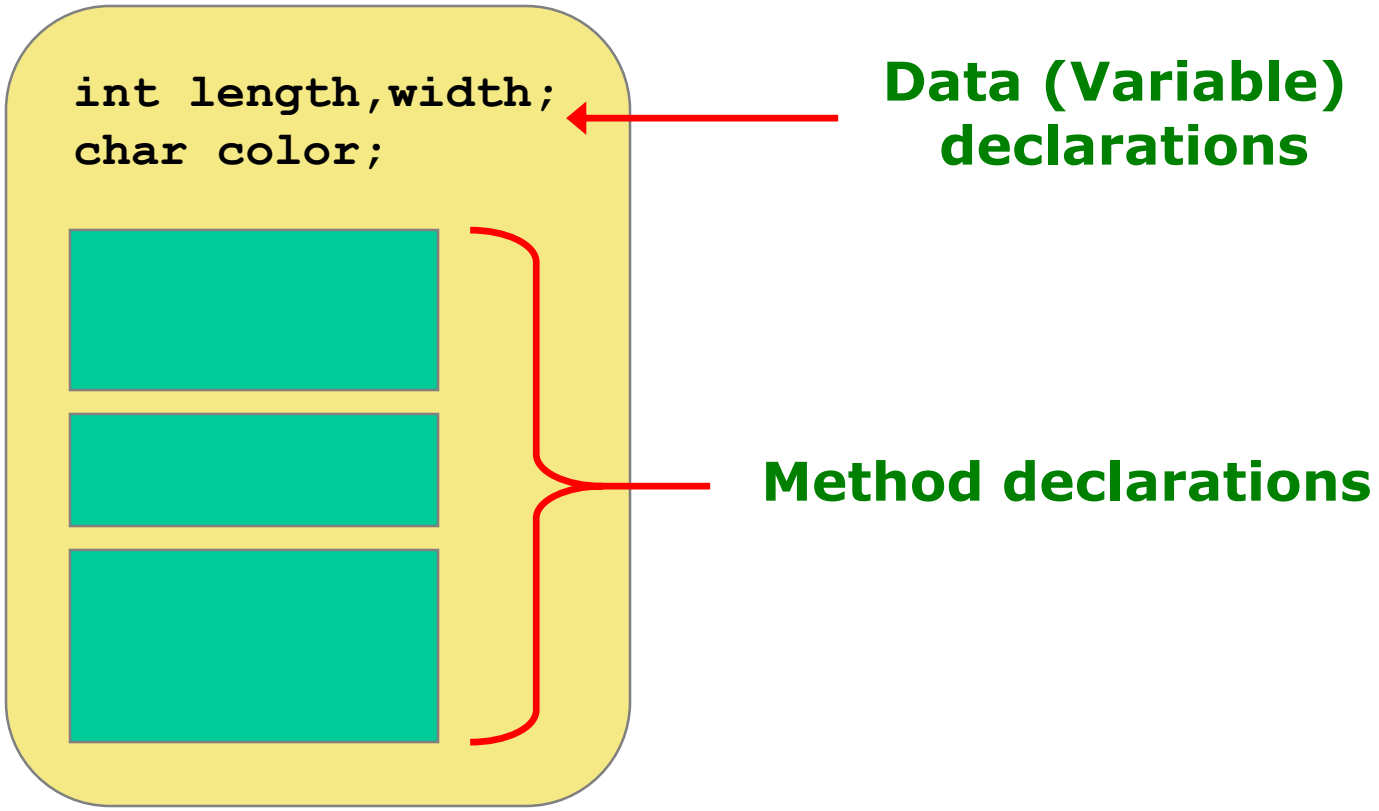
- The programs we've seen in previous examples have used classes defined in the Java standard class library
- Now we will begin to design programs that rely on classes that we write ourselves
- The class that contains the **main** method is just the starting point of a program
- True **object-**oriented programming (OOP) is based on defining classes that represent objects with well-defined characteristics (attributes) and functionality (operations)

Examples of Classes

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Classes

- A class can contain data declarations and method declarations



Classes

- The values of the data/variables define the **state** of an object created from the class
- The functionality of the methods define the **behaviors** of the object
- See the Die class (next slide)

The Die Class

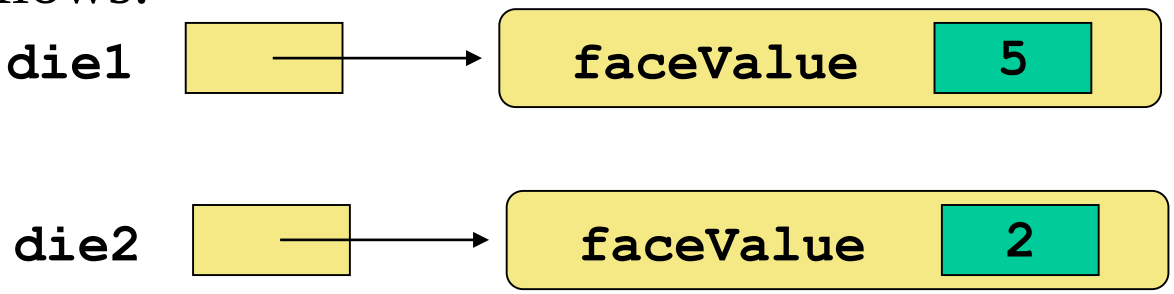
```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

faceValue that represents the current value showing on the face

continue

Instance Data

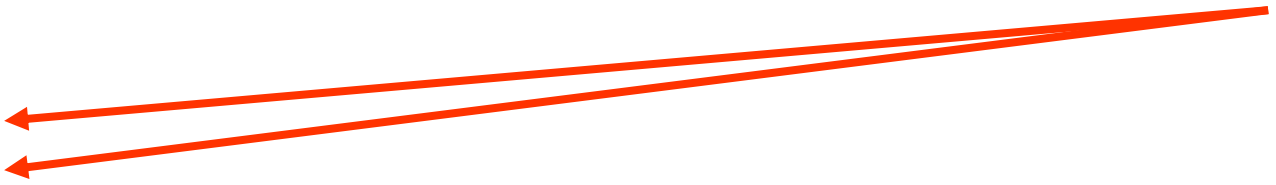
- A variable declared at the class level (such as faceValue) is called **instance data (or variable)**
- Each instance (object) has its own instance variable
- A class declares the type of the data, but it does not reserve memory space for it
- Each time a Die object is created, a new faceValue variable is created as well
- We can depict the two Die objects from the RollingDice program (next slide) as follows:



Each object maintains its own faceValue variable, and thus its own state

The RollingDice Class

```
//********************************************************************
//  RollingDice.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of a user-defined class.
//********************************************************************

public class RollingDice
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Die die1, die2;
      int sum;

      die1 = new Die();
      die2 = new Die();

      die1.roll();
      die2.roll();
      System.out.println("Die One: " + die1 + ", Die Two: " + die2);
```



continue

The Die Class

```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

continue

Constructors

- A **constructor** is used to set up an object when it is initially created
- A constructor has the same name as the class
- The Die constructor is used to set the initial face value of each new die object to one

continue

```
//-----------------------------------------------------------------
//  Rolls the die and returns the result.
//-----------------------------------------------------------------
public int roll()
{
   faceValue = (int)(Math.random() * MAX) + 1;
   return faceValue;
}

//-----------------------------------------------------------------
//  Face value mutator.
//-----------------------------------------------------------------
public void setFaceValue(int value)
{
   faceValue = value;
}

//-----------------------------------------------------------------
//  Face value accessor.
//-----------------------------------------------------------------
public int getFaceValue()
{
   return faceValue;
}
```

The methods **roll** would "roll" the die by setting faceValue to a random number between one and six

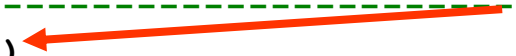
continue

```
static int abs(int num)
   Returns the absolute value of num.

static double acos(double num)

static double asin(double num)

static double atan(double num)
   Returns the arc cosine, arc sine, or arc tangent of num.

static double cos(double angle)

static double sin(double angle)

static double tan(double angle)
   Returns the angle cosine, sine, or tangent of angle, which is measured in
   radians.

static double ceil(double num)
   Returns the ceiling of num, which is the smallest whole number greater than or
   equal to num.

static double exp(double power)
   Returns the value e raised to the specified power.

static double floor(double num)
   Returns the floor of num, which is the largest whole number less than or equal
   to num.

static double pow(double num, double power)
   Returns the value num raised to the specified power.

static double random()
   Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

static double sqrt(double num)
   Returns the square root of num, which must be positive.
```

The Die Class

continue

```
   //-----------------------------------------------------------------
   //  Returns a string representation of this die.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = Integer.toString(faceValue);

      return result;
   }
}
```



The toString Method

- The toString method returns a character string that represents the object in some way
  - It's good practice to define a **toString** method for a class so that it has a printable representation
  - It is called automatically when an object is concatenated to a string or when it is passed to the **println** method (see next slide)
- It's also convenient for debugging problems

The RollingDice Class

```
//********************************************************************
//  RollingDice.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of a user-defined class.
//********************************************************************

public class RollingDice
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Die die1, die2;
      int sum;

      die1 = new Die();
      die2 = new Die();

      die1.roll();
      die2.roll();
      System.out.println("Die One: " + die1 + ", Die Two: " + die2);
```

continue

The RollingDice Class

continue

```
        die1.roll();
        die2.setFaceValue(4);
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);

        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);

        sum = die1.roll() + die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}
```

Sample Run

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```

continue

```
        die1.roll();
        die2.setFaceValue(4
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);

        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);

        sum = die1.roll() + die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}
```

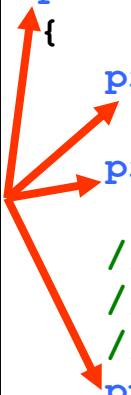
Public vs Private

- Members (variables or methods) of a class that are declared with **public** can be referenced anywhere.
- Members of a class that are declared with **private** can be referenced only within that class. (Encapsulation)
- Members declared without a modifier (default) can be referenced by any class in the same package.

Access Levels

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```



continue

The this reference

- The **this** reference allows an object to refer to itself.

```
public Account(String name, long acctNumber,
               double balance)
{
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
```

Method Overloading

- Let's look at one more important method design issue: method overloading
- **Method overloading** is the process of giving a single method name multiple definitions in a class
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The **signature** of each overloaded method must be unique
  - The signature includes the number, type, and order of the parameters

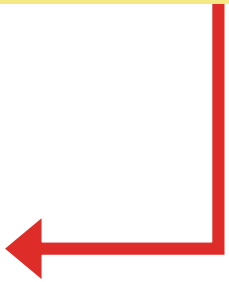
Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```



Method Overloading

- The println method is overloaded:

  println(String s)
  println(int i)
  println(double d)

  and so on...

- The following lines invoke different versions of the println method:

```
System.out.println("The total is:");
System.out.println(total);
```

Method Overloading

- In the context of method overloading, the return type of the method is not considered as part of the signature
  - That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
  - Overloaded constructors provide multiple ways to initialize a new object

Method Signatures Revisited

- A method **signature** specifies:
  - The name of the method
  - The type and name of each parameter
  - The type of the value (or object) returned by the method
  - The checked exceptions thrown by the method
  - Various method modifiers

modifiers type name (parameter list) [throws exceptions]

Static Class Members

- Recall that a static method is one that can be invoked through its class name
  - For example, the methods of the Math class are static:

result = Math.sqrt(25)

  - Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the static modifier

```
private static float price; //  a static variable

public class Helper {
   public static int cube(int num)    {// a static method
      return num * num * num;
   }
}
```

- It associates the method or variable with the class rather than with an object of that class

- Static methods are sometimes called **class methods** and static variables are sometimes called **class variables**

Static Variables

- Normally, each object has memory space for its own data
- But if a variable is declared as static:

private static float price;

- Then only one copy of the static variable exists, whose memory space is created when the class is first referenced
- All objects from the class share its static variables
  - Changing the value of a static variable in one object changes it for all others

Static Methods

```
public class Helper
{
   public static int cube(int num)
   {
      return num * num * num;
   }
}
```

- Because it is declared as static, the cube method can be invoked through the class name:

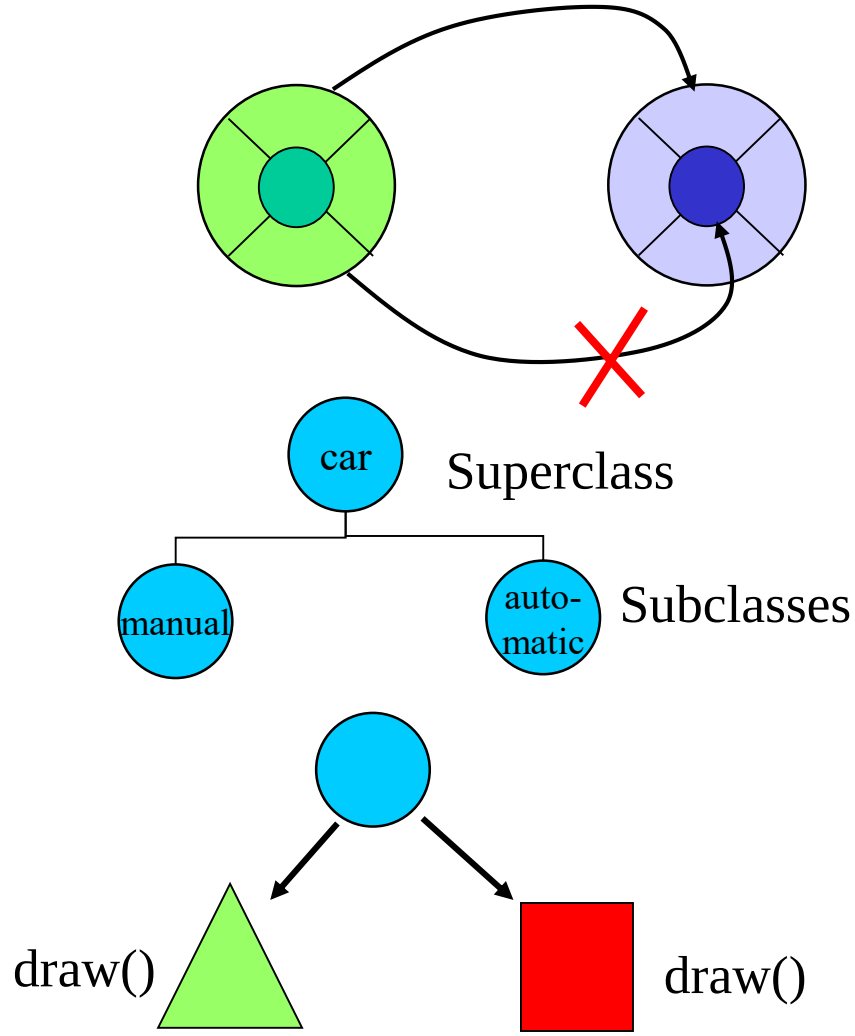
value = Helper.cube(4);

Static Methods

- Recall that the main method is static
  - It is invoked by the Java interpreter without creating an object
- Static methods **cannot** reference instance variables because instance variables don't exist until an object exists
  - However, a static method can reference static variables or local variables

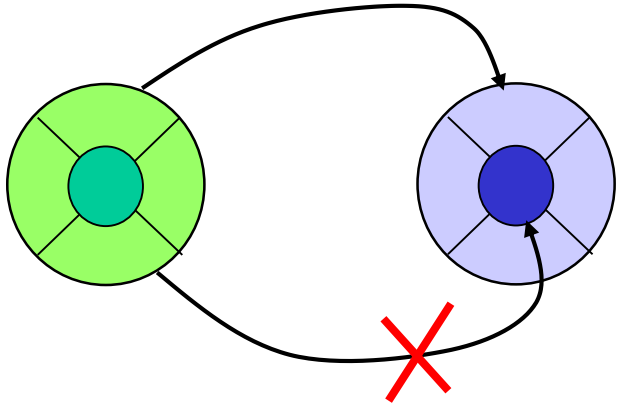
The 3 Pillars of OOP

- **Encapsulation**
  - Objects hide their functions (**methods**) and data (**instance variables**)
- **Inheritance**
  - Each **subclass** inherits all variables of its **superclass**
- **Polymorphism**
  - Same interface despite different data types



Encapsulation

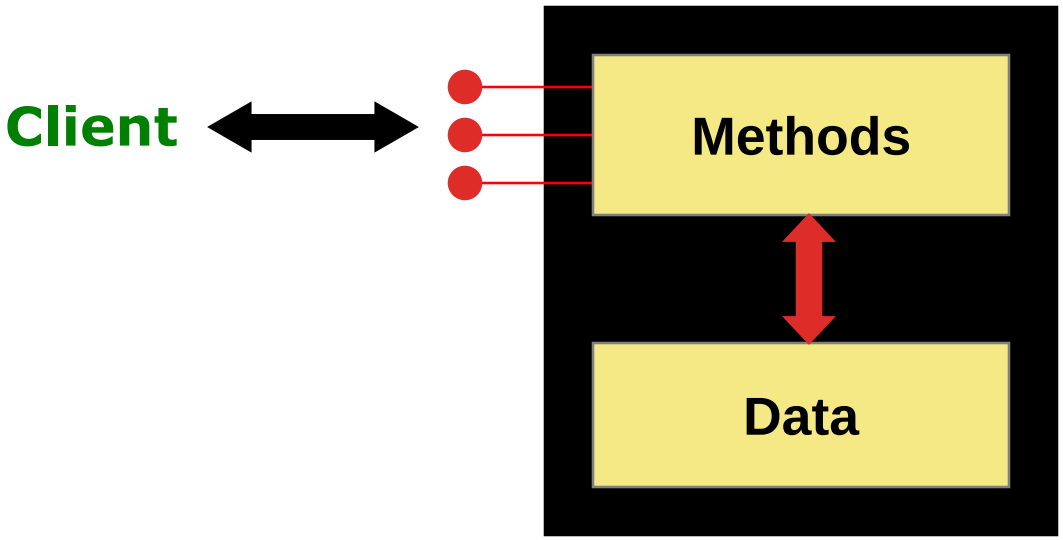
- We can take one of two views of an object:
  - **Internal** - the details of the variables and methods of the class that defines it
  - **External** - the services that an object provides and how the object interacts with the rest of the system



- From the external view, an object is an **encapsulated** entity, providing a set of specific services
  - These services define the **interface** to the object

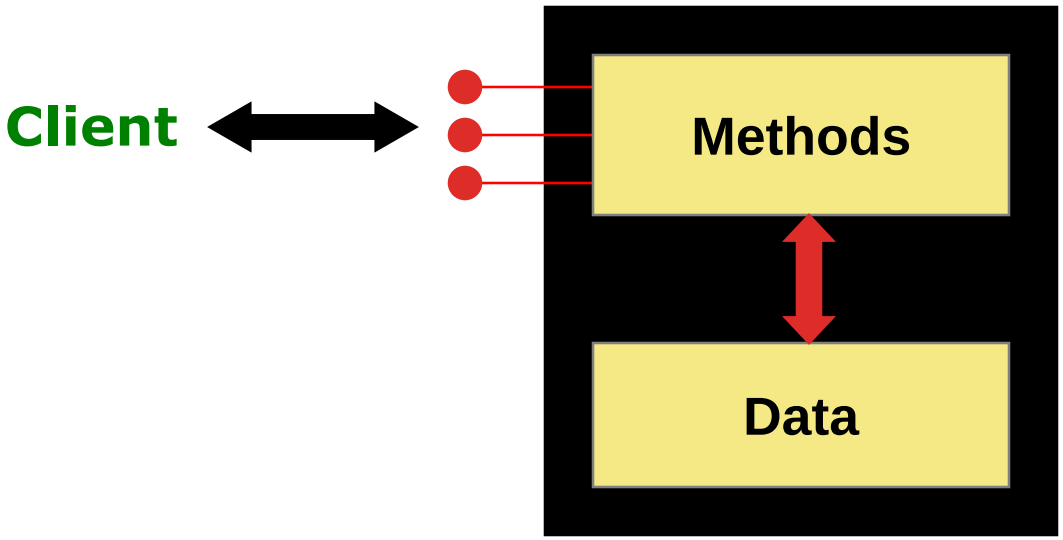
Encapsulation

- As mentioned in the previous slide, one object (called the **client**) may use another object for the services it provides, by calling its methods, but it should not have to be aware of how those services are accomplished



Encapsulation

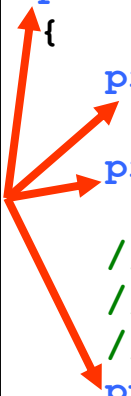
- Any changes to the object's state (its variables) should be made by that object's methods, not by the outside world (**self-governing**)
  - We should make it difficult, if not impossible, for a client to access an object's variables directly



Visibility Modifiers Revisited

- In Java, we accomplish encapsulation through the appropriate use of **visibility modifiers**
  - A modifier is a Java reserved word that specifies particular characteristics of a method or data
  - Java has three visibility modifiers: **public**, protected, and **private**
  - The protected modifier involves inheritance, which we will discuss later

```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```



continue

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to modify the values directly
  - Therefore instance variables should not be declared with public visibility
  - It is acceptable to give a **constant** public visibility, which allows it to be used outside of the class
    - Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
  - Public methods are also called **service methods**
- A method created simply to assist a service method is called a **support method**
  - Since a support method is not intended to be called by a client, it should not be declared with public visibility

Visibility Modifiers

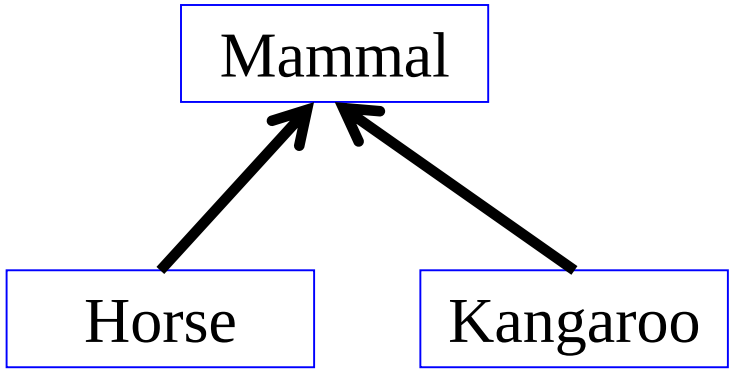
	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- Because instance variables are private, a class usually provides services to access and modify data values (via **service methods):**
  - An **accessor method** returns the current value of a variable
  - A **mutator method** changes the value of a variable
- The names of accessor and mutator methods take the form getX and setX, respectively, where X is the name of the value
- They are sometimes called "**getters**" and "**setters**"

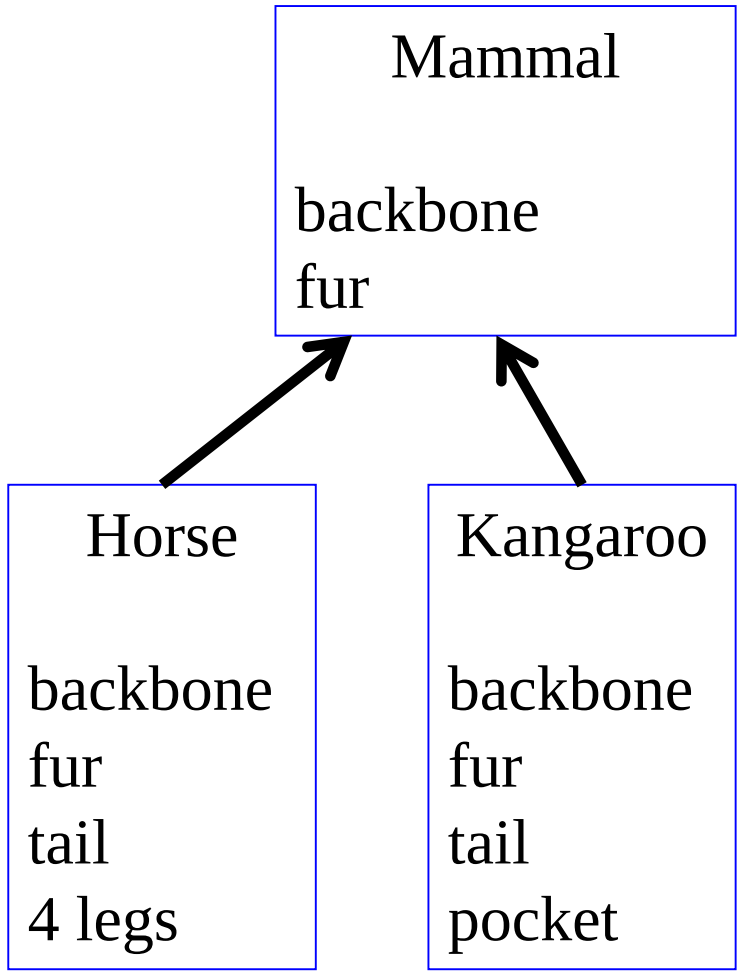
Inheritance

- **Inheritance** allows a software developer to derive a new class from an existing one
- The existing class is called the **parent class**, or **superclass**, or **base class**
- The derived class is called the **child class** or **subclass**



Inheritance

- As the name implies, the child inherits characteristics of the parent
  - That is, the child class inherits the **methods** and **data** defined by the parent class



Deriving Subclasses

- In Java, we use the reserved word **extends** to establish an inheritance relationship

```
public class Duck extends Animal
{
    // class contents
}
```

Base Class: Animal

```
//Animal.java  base class
public class Animal {
   //instance data/variables
   protected String name;
   protected boolean flies;
```

The protected Modifier

- The **protected** modifier allows a child class to reference a variable or method in the parent class
  - It provides more encapsulation than **public** visibility, but is not as tightly encapsulated as **private** visibility
    - Variables and methods declared with **private** visibility cannot be referenced in a child class
    - They can be referenced in the child class if they are declared with **public** visibility -- but public variables violate the principle of encapsulation
  - A protected variable is also visible to any class in the same package as the parent class

Access Levels

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Base Class: Animal

```
//constructors
public Animal()
{ name = "just an animal";
  flies = false;}

public Animal(String n)
{ name = n;
  flies = false;}
```

Constructors can be overloaded. Overloaded constructors provide multiple ways to initialize a new object

Base Class: Animal

```
  //other methods
  public String toString()
  { return name; }

  public void talk()
  { System.out.println("Indistinct animal noises.");}

  public void eat()
  { System.out.println("Eat eat eat.");  }

}//end Animal
```

child Class: Duck

```
//Duck.java
public class Duck extends Animal {
   //instance data
   protected int numberOfFeathers;
```

Child Class: Duck

```
//constructors
public Duck()
{  super("duck");
   flies = true;
   numberOfFeathers=1000; }

public Duck(String n, int f )
{   super(n);
    flies = true;
    numberOfFeathers = f; }
```


178

The super Reference

- Constructors are not inherited, even though they have public visibility
  - The super reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the super reference to call the parent's constructor
- The super reference can also be used to reference other variables and methods defined in the parent's class. (see next slide: `super.toString()`)

Child Class: Duck

```
//other methods
 public String toString()
 {
   String result = "I am a ";
   result += super.toString() + ".\n";
   result += "I can" + (flies ? "" : "not")
             + " fly.\n";
   result += "I have " + numberOfFeathers
             + " feathers.";
   return result;
 }
```

Child Class: Duck

```
  public void talk()
  { System.out.println("Quack! Quack!"); }

  public void move()
  { System.out.println("I swim.  I fly. "
                       +  "I walk."); }

  public void breakWing()
  { flies = false; }

}//end Duck
```

Overriding Methods

- A child class can **override** the definition of an inherited method in favor of its own
  - The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
  - Examples from AnimalTester
- A method in the parent class can be invoked explicitly using the super reference
  - If a method is declared with the **final** modifier, it cannot be overridden
- A child class can also override variables in the parent. (uncommon)

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different parameters
- Overriding lets you define a similar operation in different ways for different object types

Driver Class: AnimalTester

```
//AnimalTester.java
public class AnimalTester {
 public static void main(String[] args)
  {
  Animal ginger = new Animal("cat");
  System.out.println(ginger);
  ginger.talk();
  ginger.eat();
  System.out.println();
```

driver class: AnimalTester

```
  Duck daffy = new Duck("mallard", 12000);
  System.out.println(daffy);
  daffy.talk();
  daffy.eat();
  daffy.move();
  daffy.breakWing();
  System.out.println(daffy);
  } //end main
}
```

Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.

A child class can override the constructor of the parent class.

A child class cannot override a final method of the parent class.

It is considered poor design when a child class overrides a method from the parent.

A child class may define a variable with the same name as a variable in the parent.

Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.	True
A child class can override the constructor of the parent class.	False
A child class cannot override a final method of the parent class.	True
It is considered poor design when a child class overrides a method from the parent.	False
A child class may define a variable with the same name as a variable in the parent.	True, but shouldn't

Polymorphism

- The term **polymorphism** literally means "having many forms“
- A **polymorphic reference** is a variable that can refer to different types of objects at different points in time
- Suppose we create the following reference variable:

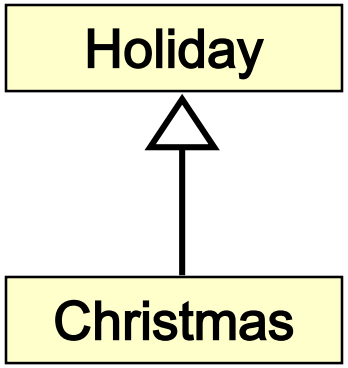
Holiday day;

- This reference can point to an **Holiday** object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces

References and Inheritance

- For example, if Holiday is the superclass of Christmas, then a Holiday reference could be used to refer to a Christmas object

- These type compatibility rules are just an extension of the **is-a** relationship established by inheritance




```
Holiday day;
day = new Christmas();
```

References and Inheritance

- Assigning a Christmas object to a Holiday reference is fine because Christmas **is-a** holiday
  - Assigning a child object to a parent reference can be performed by simple assignment
  - Assigning an parent object to a child reference can be done also, but must be done with a **cast:**

```
Holiday day = new Christmas();
Christmas today;
today = (Christmas) day;
```

References and Inheritance

- Now suppose the Holiday class has a method called **celebrate**, and Christmas overrides it
- What method is invoked by the following?

day.celebrate();

  - The type of the object being referenced, not the reference type, determines which method is invoked
  - If day refers to a Holiday object, it invokes the Holiday version of celebrate
  - If it refers to a Christmas object, it invokes that version

Interfaces

- High-level description of a class behavior
- Listing of method signatures
- Interfaces are defined with the following syntax:

[*visibility*] interface ***InterfaceName*** [extends *other interfaces*] {

constant declarations

abstract method declarations

}

- The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the abstract keyword is not required.
- Since the interface specifies a set of exposed behaviors, all methods are implicitly public.

Interfaces

- A bicycle's behavior, if specified as an interface, might appear as follows:

```
public interface Bicycle {
    // wheel revolutions per minute
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

- To implement this interface:

```
public class ACMEBicycle implements Bicycle {
    // remainder of this class
}
```

Quick Check

If MusicPlayer is the parent of CDPlayer, are the following assignments valid?

MusicPlayer mplayer = new CDPlayer();

CDPlayer cdplayer = new MusicPlayer();

Quick Check

If MusicPlayer is the parent of CDPlayer, are the following assignments valid?

MusicPlayer mplayer = new CDPlayer();

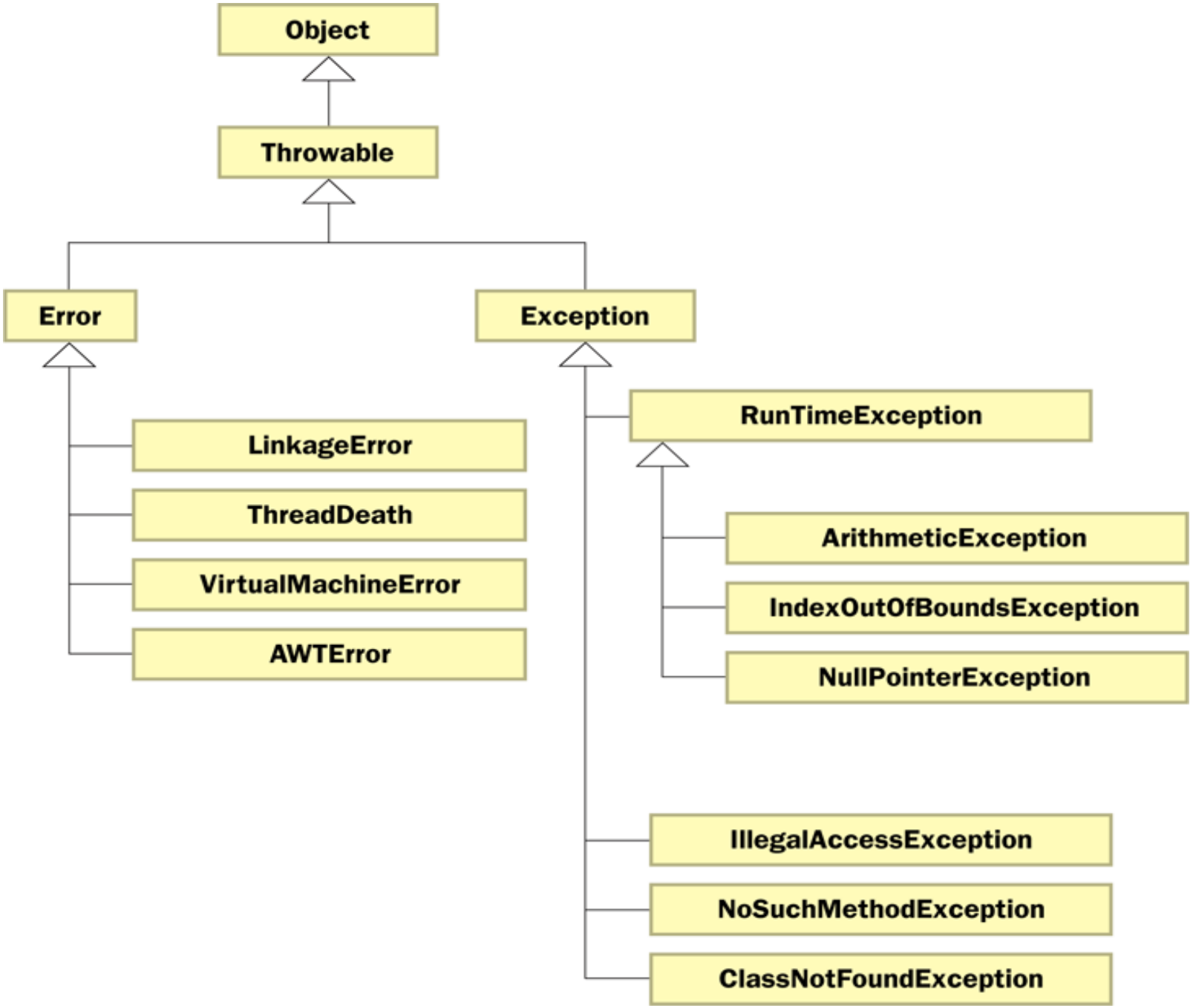
Yes, because a CDPlayer is-a MusicPlayer

CDPlayer cdplayer = new MusicPlayer();

No, you'd have to use a cast (and you shouldn't knowingly assign a super class object to a subclass reference)

Exceptions

- An **exception** is an object that describes an unusual or erroneous situation
- Exceptions are **thrown** by a program, and may be **caught** and **handled** by another part of the program
- A program can be separated into a normal execution flow and an **exception execution flow**
- An **error** is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught. (see next slide)


Object
Throwable
Error
Exception
LinkageError
ThreadDeath
VirtualMachineError
AWTError
RunTimeException
ArithmeticException
IndexOutOfBoundsException
NullPointerException
IllegalAccessException
NoSuchMethodException
ClassNotFoundException

Exception Handling

- The Java API has a predefined set of exceptions that can occur during execution
- A program can deal with an exception in one of three ways:
  - **Ignore it**
  - Handle it where it occurs
  - Handle it an another place in the program
- The manner in which an exception is processed is an important design consideration

Exception Handling

- If an exception is ignored (not caught) by the program, the program will terminate and produce an appropriate message
- The message includes a **call stack trace** that:
  - Indicates the line on which the exception occurred
  - Shows the method call trail that lead to the attempted execution of the offending line
- See Zero.java

```
//********************************************************************
//  Zero.java       Author: Lewis/Loftus
//
//  Demonstrates an uncaught exception.
//********************************************************************

public class Zero
{
   //-----------------------------------------------------------------
   //  Deliberately divides by zero to produce an exception.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int numerator = 10;
      int denominator = 0;

      System.out.println(numerator / denominator);

      System.out.println("This text will not be printed.");
   }
}
```

Output (when program terminates)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Zero.java:17)
```

```
//
//  Zero.java       Author: Lewis/Loftus
//
//  Demonstrates an uncaught exception.
//********************************************************************

public class Zero
{
   //-----------------------------------------------------------------
   //  Deliberately divides by zero to produce an exception.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int numerator = 10;
      int denominator = 0;

      System.out.println(numerator / denominator);

      System.out.println("This text will not be printed.");
   }
}
```

Exception Handling

- The Java API has a predefined set of exceptions that can occur during execution
- A program can deal with an exception in one of three ways:
  - Ignore it
  - **Handle it where it occurs**
  - Handle it an another place in the program
- The manner in which an exception is processed is an important design consideration

The try Statement

- To handle an exception in a program, use a **try-catch statement**
- A **try block** is followed by one or more **catch** clauses
- Each catch clause has an associated exception type and is called an **exception handler**
- When an exception occurs within the try block, processing immediately jumps to the first catch clause that matches the exception type
- See ProductCodes.java

```
//********************************************************************
//  ProductCodes.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a try-catch block.
//********************************************************************

import java.util.Scanner;

public class ProductCodes
{
   //-----------------------------------------------------------------
   //  Counts the number of product codes that are entered with a
   //  zone of R and and district greater than 2000.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String code;
      char zone;
      int district, valid = 0, banned = 0;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter product code (XXX to quit): ");
      code = scan.nextLine();
```

continue

continue

```
      while (!code.equals("XXX"))
      {
         try
         {
            zone = code.charAt(9);
            district = Integer.parseInt(code.substring(3, 7));
            valid++;
            if (zone == 'R' && district > 2000)
               banned++;
         }
         catch (StringIndexOutOfBoundsException exception)
         {
            System.out.println("Improper code length: " + code);
         }
         catch (NumberFormatException exception)
         {
            System.out.println("District is not numeric: " + code);
         }

         System.out.print("Enter product code (XXX to quit): ");
         code = scan.nextLine();
      }

      System.out.println("# of valid codes entered: " + valid);
      System.out.println("# of banned codes entered: " + banned);
   }
}
```

continue

```
      while
      {
         try
         {
            }
         cat
         {
            }
         catch (NumberFormatException exception)
         {
            System.out.println("District is not numeric: " + code);
         }

         System.out.print("Enter product code (XXX to quit): ");
         code = scan.nextLine();
      }

      System.out.println("# of valid codes entered: " + valid);
      System.out.println("# of banned codes entered: " + banned);
   }
}
```

Sample Run

```
Enter product code (XXX to quit): TRV2475A5R-14
Enter product code (XXX to quit): TRD1704A7R-12
Enter product code (XXX to quit): TRL2k74A5R-11
District is not numeric: TRL2k74A5R-11
Enter product code (XXX to quit): TRQ2949A6M-04
Enter product code (XXX to quit): TRV2105A2
Improper code length: TRV2105A2
Enter product code (XXX to quit): TRQ2778A7R-19
Enter product code (XXX to quit): XXX
# of valid codes entered: 4
# of banned codes entered: 2
```

The finally Clause

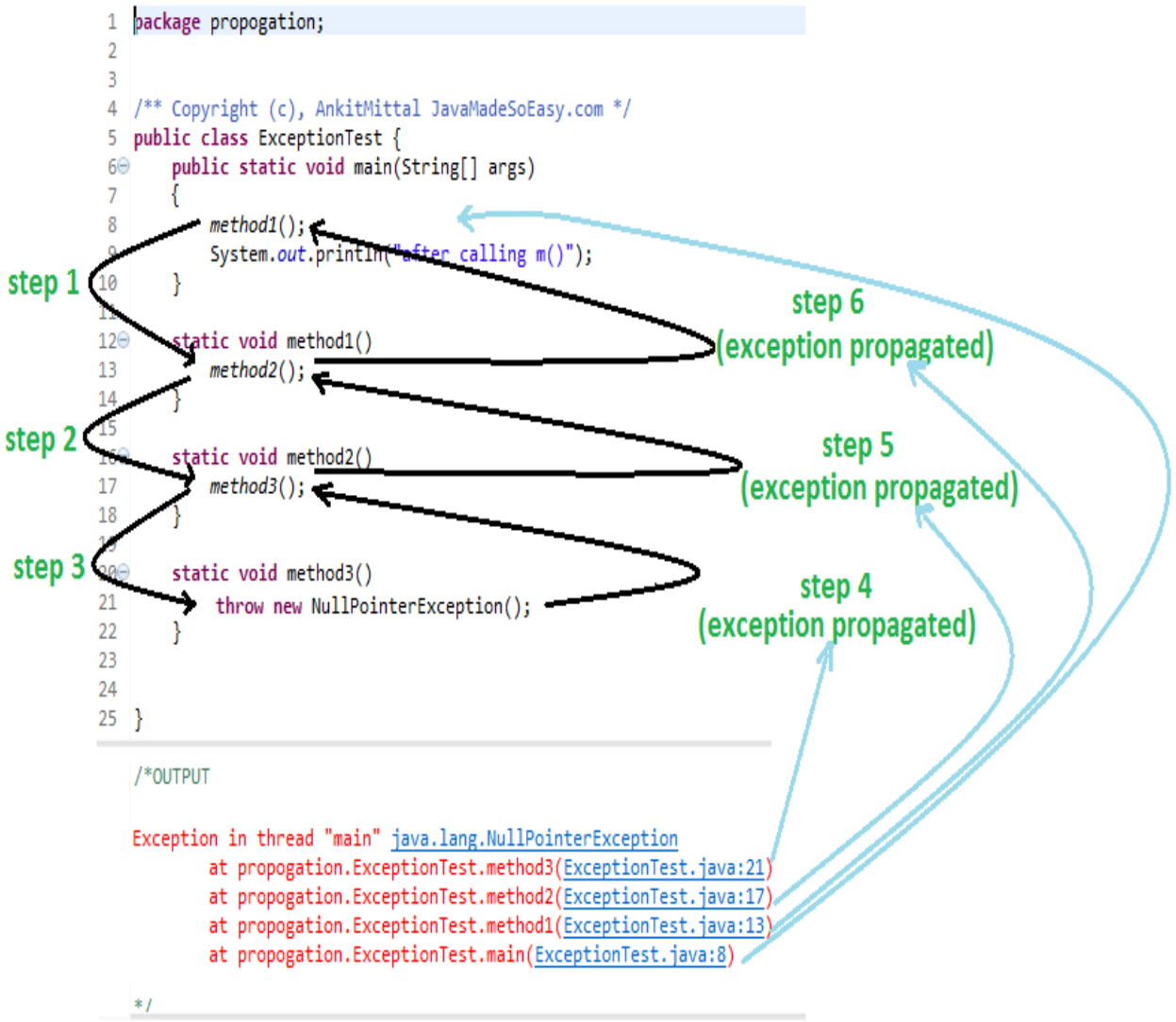
- A try statement can have an optional finally clause, which is always executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block finish
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause finish

Exception Handling

- The Java API has a predefined set of exceptions that can occur during execution
- A program can deal with an exception in one of three ways:
  - Ignore it
  - Handle it where it occurs
  - **Handle it an another place in the program**
- The manner in which an exception is processed is an important design consideration

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions **propagate** up through the method calling hierarchy until they are caught and handled or until they reach the level of the main method

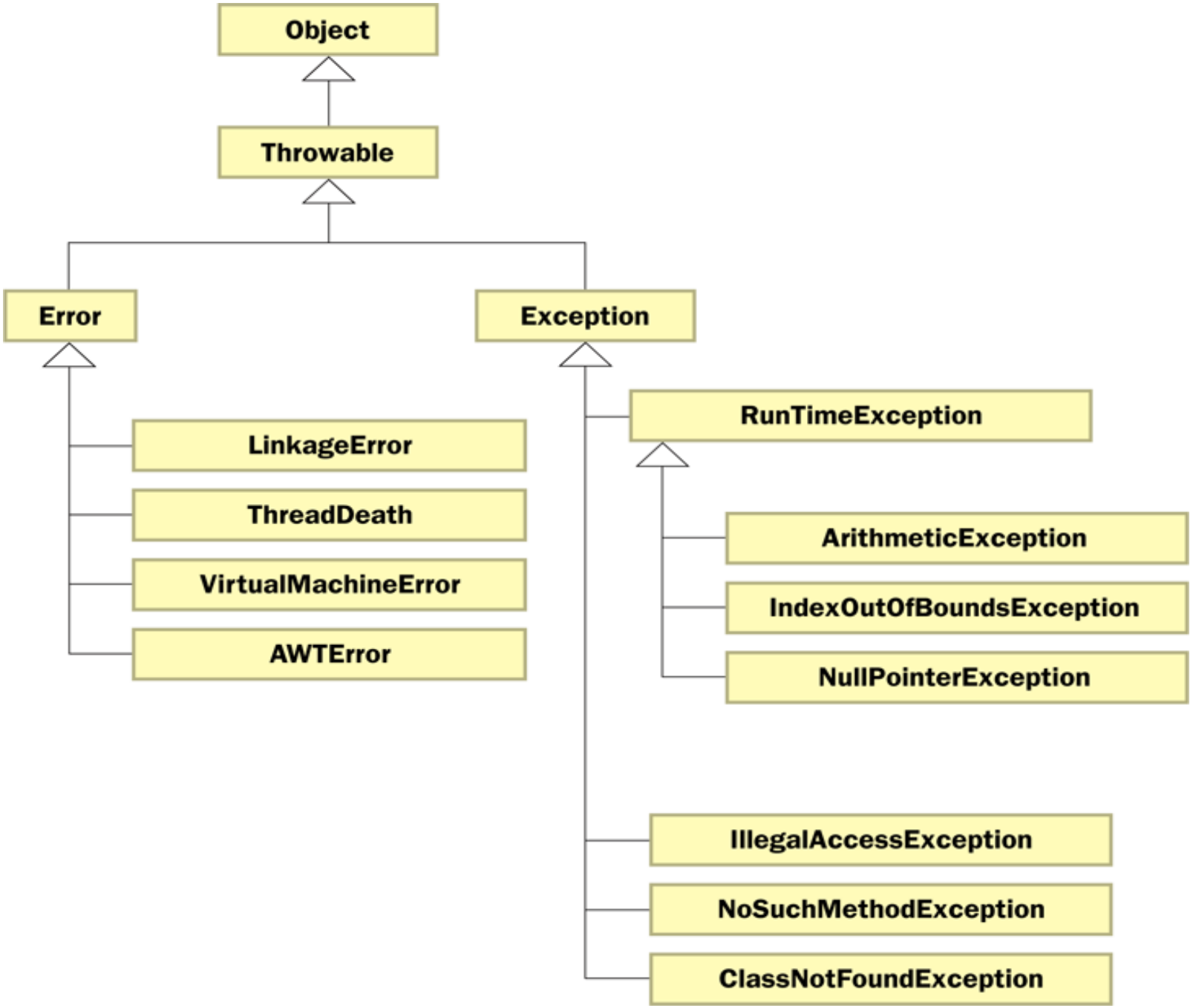



```
1  package propogation;
2
3
4  /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
5  public class ExceptionTest {
6      public static void main(String[] args)
7      {
8          method1();
9          System.out.println("after calling m()");
10     }
11
12     static void method1()
13         method2();
14     }
15
16     static void method2()
17         method3();
18     }
19
20     static void method3()
21         throw new NullPointerException();
22     }
23
24
25 }
```

```
/*OUTPUT

Exception in thread "main" java.lang.NullPointerException
        at propogation.ExceptionTest.method3(ExceptionTest.java:21)
        at propogation.ExceptionTest.method2(ExceptionTest.java:17)
        at propogation.ExceptionTest.method1(ExceptionTest.java:13)
        at propogation.ExceptionTest.main(ExceptionTest.java:8)

*/
```

The Exception Class Hierarchy

- Exception classes in the Java API are related by inheritance, forming an exception class hierarchy (next slide)
- All error and exception classes are descendents of the **Throwable** class
- A programmer can define an exception by extending the Exception class or one of its descendants
- The parent class used depends on how the new exception will be used


Object
Throwable
Error
Exception
LinkageError
ThreadDeath
VirtualMachineError
AWTError
RunTimeException
ArithmeticException
IndexOutOfBoundsException
NullPointerException
IllegalAccessException
NoSuchMethodException
ClassNotFoundException

The throw Statement

- Exceptions are thrown using the **throw** statement
- Usually a throw statement is executed inside an **if statement** that evaluates a condition to see if the exception should be thrown
- See CreatingExceptions.java
- See OutOfRangeException.java

```
//********************************************************************
//  OutOfRangeException.java       Author: Lewis/Loftus
//
//  Represents an exceptional condition in which a value is out of
//  some particular range.
//********************************************************************

public class OutOfRangeException extends Exception
{
   //-----------------------------------------------------------------
   //  Sets up the exception object with a particular message.
   //-----------------------------------------------------------------
   OutOfRangeException(String message)
   {
      super(message);
   }
}
```

```
//********************************************************************
//  CreatingExceptions.java       Author: Lewis/Loftus
//
//  Demonstrates the ability to define an exception via inheritance.
//********************************************************************

import java.util.Scanner;

public class CreatingExceptions
{
   //-----------------------------------------------------------------
   //  Creates an exception object and possibly throws it.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws OutOfRangeException
   {
      final int MIN = 25, MAX = 40;

      Scanner scan = new Scanner(System.in);

      OutOfRangeException problem =
         new OutOfRangeException("Input value is out of range.");
```

continue

continue

```
      System.out.print("Enter an integer value between " + MIN +
                       " and " + MAX + ", inclusive: ");
      int value = scan.nextInt();

      //  Determine if the exception should be thrown
      if (value < MIN || value > MAX)
         throw problem;

      System.out.println("End of main method.");  // may never reach
   }
}
```

Sample Run

```
Enter an integer value between 25 and 40, inclusive: 69
Exception in thread "main" OutOfRangeException:
    Input value is out of range.
    at CreatingExceptions.main(CreatingExceptions.java:20)
```

continu

```
        System.out.print("Enter an integer value between " + MIN +
                         " and " + MAX + ", inclusive: ");
        int value = scan.nextInt();

        //  Determine if the exception should be thrown
        if (value < MIN || value > MAX)
           throw problem;

        System.out.println("End of main method.");  // may never reach
    }
}
```

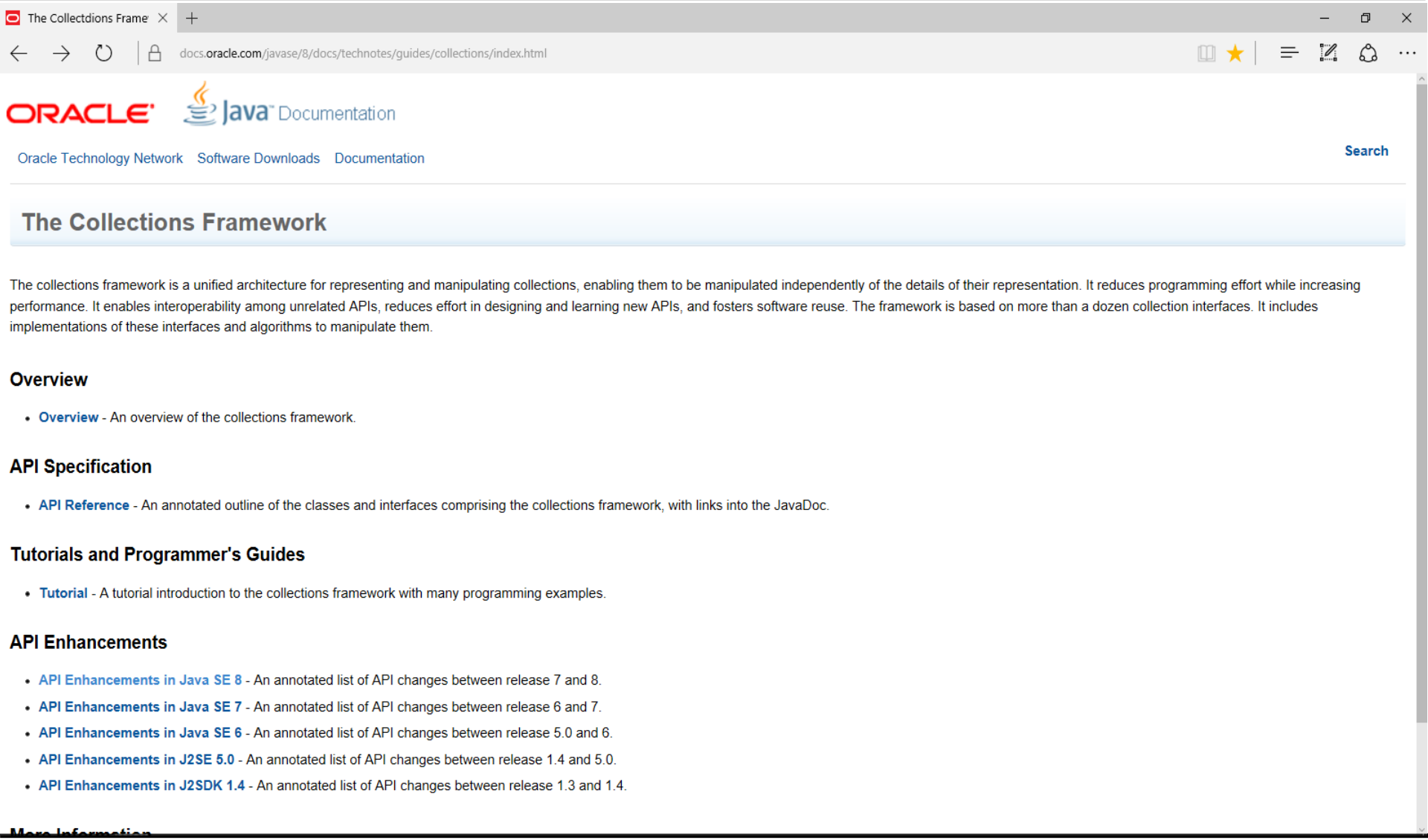
Collections

- A **collection** is an object that serves as a repository for other objects
- A collection provides services for adding, removing, and managing the elements it contains
- Sometimes the elements in a collection are ordered, sometimes they are not
- Sometimes collections are **homogeneous**, containing all the same type of objects, and sometimes they are **heterogeneous**

Collection Classes

- The Java standard library contains several classes that represent collections, often referred to as the **Java Collections API**
- Their underlying implementation is implied in the class names such as **ArrayList** and **LinkedList**

The Java Collections API



https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html

The ArrayList Class

- An array list automatically **resizes** itself whenever necessary.
- An array list lets you **insert** elements into the middle of the collection.
- An array list lets you **delete** items.
- The ArrayList class actually uses an **array** internally to store the data you add to the array list.

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Direct Known Subclasses:

`AttributeList, RoleList, RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant

Creating an ArrayList Object

```
import java.util.ArrayList // or java.util.*.

ArrayList signs;
signs = new ArrayList();
```

Or

```
ArrayList signs = new ArrayList();
```

Or

```
ArrayList signs = new ArrayList(100); // initial capacity, default =10
```

Or

```
ArrayList<String> signs = new ArrayList<String>(); //generics
```

Adding/Accessing Elements

signs.**add**(“Drink Pepsi”);
signs.**add**(“No minors allowed”);
signs.**add**(“Say Pepsi, Please”);

signs.**add**(2, “7-Up: You Like It, It Likes You”);

```
for (int i = 0; i < signs.size(); i++)
    System.out.println(signs.get(i)); // use getter
```

Or,

```
ArrayList<String> signs = new ArrayList<String>(); //required
for (String s : signs) // enhanced for loop
    System.out.println(s);
```

Updating/Deleting Elements

```
//Updating elements
signs.set(0, “Drink Coke”); // use setter

// Deleting elements
signs.clear(); // remove all the elements
signs.remove(0); // remove a specific element
```

The LinkedList Class

- A **linked list** is a collection in which every object in the list maintains with it a pointer to the next object in the list and the previous object in the list.
- No array is involved at all in a linked list.
- Instead, the list is managed entirely by these pointers.
- Linked lists are especially well suited for creating two common types of lists:
  - **Stacks:** A stack is a list in which items can only be added to and retrieved from the front of the list.
  - **Queues:** A queue is a list in which items are always added to the back of the list and always retrieved from the front.

compact1, compact2, compact3

java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>`

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be

The LinkedList Class

```
import java.util.LinkedList // or java.util.*.

LinkedList signs = new LinkedList();
```

Or

```
LinkedList<String> signs = new LinkedList<String>(); //generics
```

The LinkedList Class

```
signs.add("Drink Pepsi");
signs.add("No minors allowed");
signs.add("Say Pepsi, Please");

signs.add(2, "7-Up: You Like It, It Likes You");

for (int i = 0; i < signs.size(); i++)
    System.out.println(signs.get(i)); // use getter
Or,
LinkedList<String> signs = new LinkedList<String>(); //required
for (String s : signs) // enhanced for loop
    System.out.println(s);
```

The LinkedList Class

```
//Updating elements
signs.set(0, “Drink Coke”); // use setter

// Deleting elements
signs.clear(); // remove all the elements
signs.remove(0); // remove a specific element
```

Stream-Based I/O in Java

- I/O in Java is built on **streams**.
  - Input streams read data; output streams write data.
  - Different stream classes, such as **java.io.FileInputStream**, read and write particular sources of data.
  - All output streams have the same basic methods to write data.
  - All input streams use the same basic methods to read data.
  - To be discussed in detail later when we get to Chapter 3 of Textbook #1 (TCP/IP Sockets in Java)

Garbage Collection

- The **new** operator explicitly creates objects
- No way to explicitly free objects
- Java VM automatically identifies unused objects and frees them during periods of low activity
- Can use **finalize** method to guide it (not commonly used)

```
class MyClass {
   public void cleanup() { ... }
   protected void finalize() throws Throwable  {
      if (!cleanedUp)  { cleanup(); }
      super.finalize();
   }
}
```

- Can manually call the garbage collector

```
System.gc();
```

java.lang

Class Object

java.lang.Object

---

public class **Object**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

`JDK1.0`

See Also:

`Class`

Constructor Summary

Constructors

Constructor and Description
`Object()`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
protected `Object`	`clone()` Creates and returns a copy of this object.
boolean	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
protected void	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.

Packages

Java	C++
package *Package*;	namespace *Namespace*{...}
which must appear as the first line in the file	

Packages

- Java code has hierarchical structure.
  - The environment variable CLASSPATH contains the directory names of the roots.
  - Every class belongs to a **package** (It's a keyword!).
- Classes with no package declarations are implicitly part of an "**unnamed package**", often also called "**default package**".
  - It's not possible to import classes from an unnamed package
- An class's full (fully-qualified) name includes the name of the package containing it.

  packageName.className

Packages

To create a package:

```
package p1;
public class c1 {
        public void m1() {
                System.out.println("Method m1 of Class c1");
        }
        public static void main(String args[]) {
                c1 obj = new c1();
                obj.m1();
        }
}
```

❖Compile the file as: (the . after –d refers to the current directory)

javac –d . c1.java

❖Run the code as:

java p1.c1

Packages

To use a package you have created:

```
// Using the packages created earlier
package p2;
import p1.*; //imports classes in package p1
public class c2 {
        public   void m2(){
                System.out.println("Method m2 of Class c2");
        }
        public static void main(String args[]){
                c1 obj1 = new c1();
                obj1.m1();
                c2 obj2 = new c2();
                obj2.m2()
        }
}
```

Packages

To import a package:

```
JFrame f = new javax.swing.JFrame; // use fully qualified name
```

❖Or

```
import javax.swing.JFrame // here only the JFrame class is imported
JFrame f = new JFrame; // without fully qualified name
```

❖Or

```
import javax.swing.*; // * import all classes in this package
JFrame f = new JFrame; // without fully qualified name
```

Reading Assignment

- The Java Tutorials at Oracle.com:
  http://docs.oracle.com/javase/tutorial/
- At a minimum:
  - All sections under the **Getting Started** trail
  - All sections under **Learning the Java Language** trail
  - The **Exceptions** section under the **Essential Classes** trail
- If you have taken ECS 102 the Java version, then you should be quite familiar with those topics already.