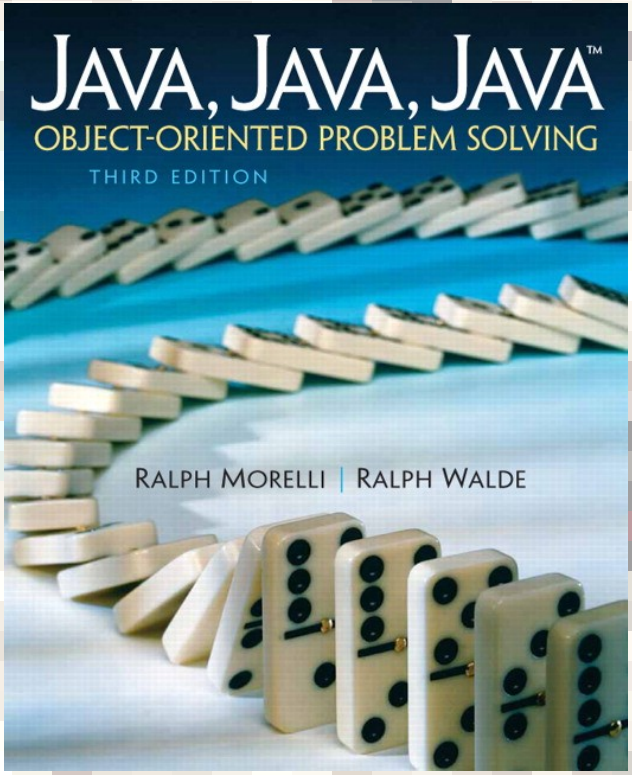


presentation slides for

JAVA, JAVA, JAVA

Object-Oriented Problem Solving

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Introduction – Course Logistics

Course Logistics

- Course Websites
  - D2L
  - Online
- Textbook

  Java, Java, Java, 3E by R. Morelli | R. Walde – open source freely available book

- Class Hours

  MW, 10:00 AM to 11:50 AM (Final Exam Dec 8)

- Office Hours:
  - TH 1:00 PM - 4:00 PM (Zoom link on the homepage of D2L)
  - By appointment.
- Grading
  - 20% Assignments, 15% Labs, 30% Test 1 and 2, 15% Final Project, and 25 % Final exam
  - Fixed grading scale (90% A, 80% B, 70% C, 60% D, < 60% F)
- More Detailed: Read Syllabus

Java, Java, Java

Object Oriented Problem Solving

Review CS234

Why Study Java?

- Java is *Object Oriented (OO)*: dividing programs into modules (*objects*) helps manage software complexity.
- Java is *robust*: errors in Java don't usually cause system crashes as in other languages.
- Java is *platform independent*: programs run without changes on all different platforms.
- Java is a *distributed* language: programs can easily be run on computer networks.
- Java is a *secure* language: it guards against viruses and other untrusted code.

What is an Object?

- An ***object*** is any thing whatsoever. It can be physical (Car), mental (Idea), natural (Animal), artificial (ATM).
- We use *UML* (Unified Modeling Language) to depict objects.

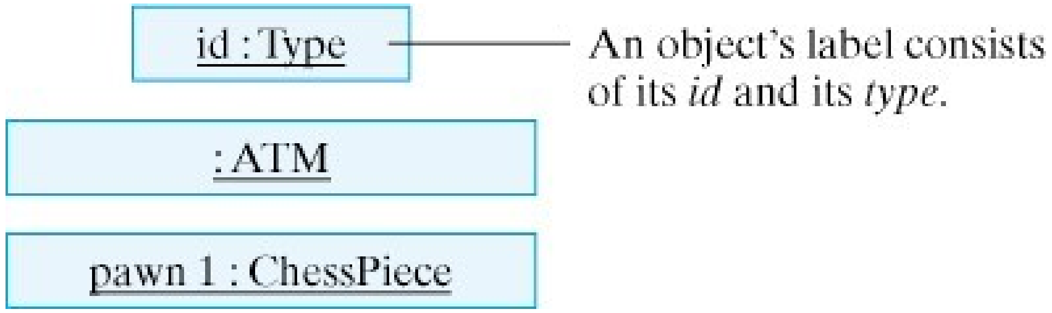


Figure 0.5. In UML objects are represented by rectangles Labeled with a two part label of the form *id:Type.*

What is a Java Class?

- A ***class*** (Rectangle) is a ***blueprint*** or ***template*** of all objects of a certain type.
- An object is an ***instance*** of a class.

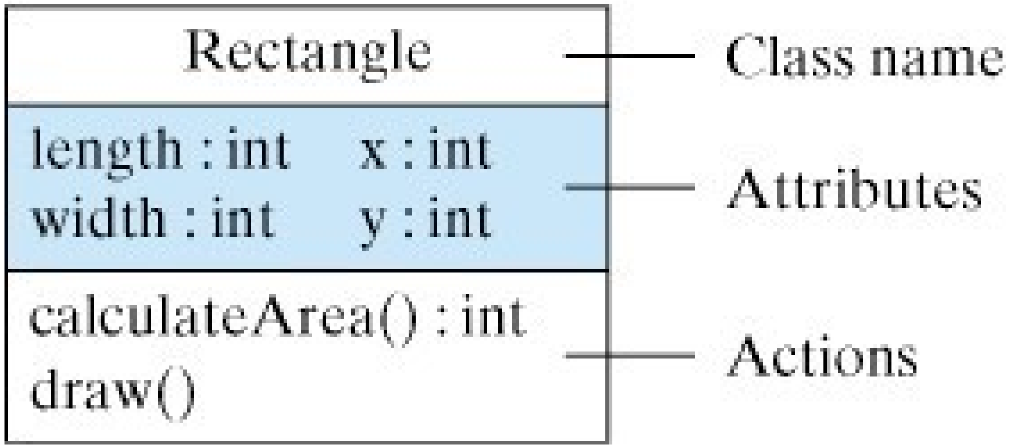
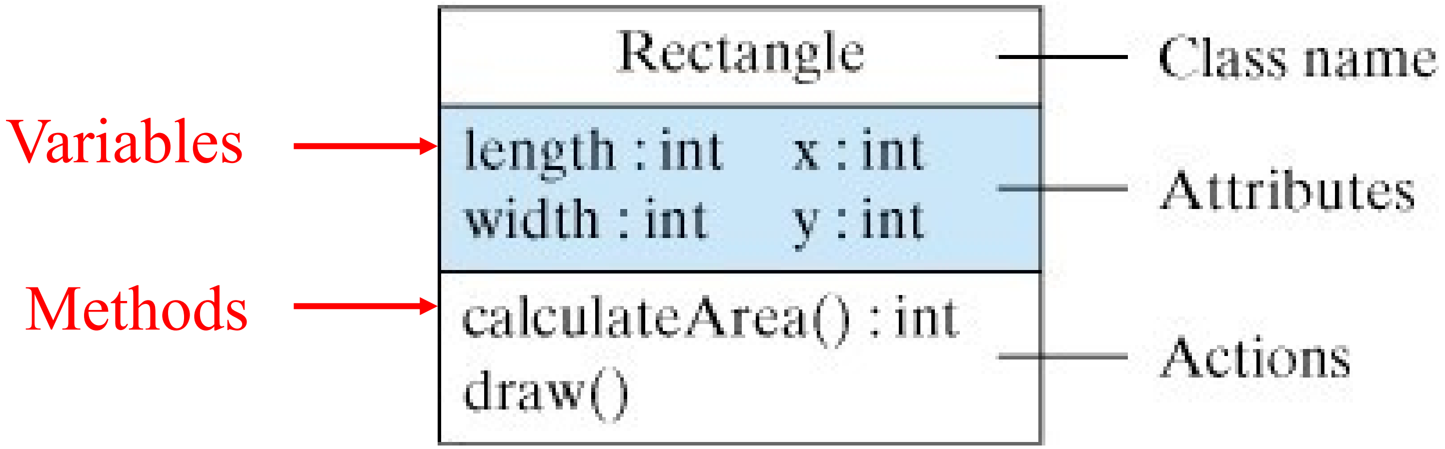


Figure 0.9. A UML Class Diagram

Variables and Methods

- A *variable*, which corresponds to an attribute, is a named memory location that can store a certain type of value.
- A *method*, which corresponds to an action, is a named chunk of code that can be *invoked* to perform a certain predefined set of actions.



Instance vs. Class Variables and Methods

- Question: What are instance variable and class variable?

Instance vs. Class Variables and Methods

- A *instance variable* (or *instance method*) is a variable or method that belongs to an object.
- A *class variable* (or *class method*) is a variable or method that is associated with the class itself.

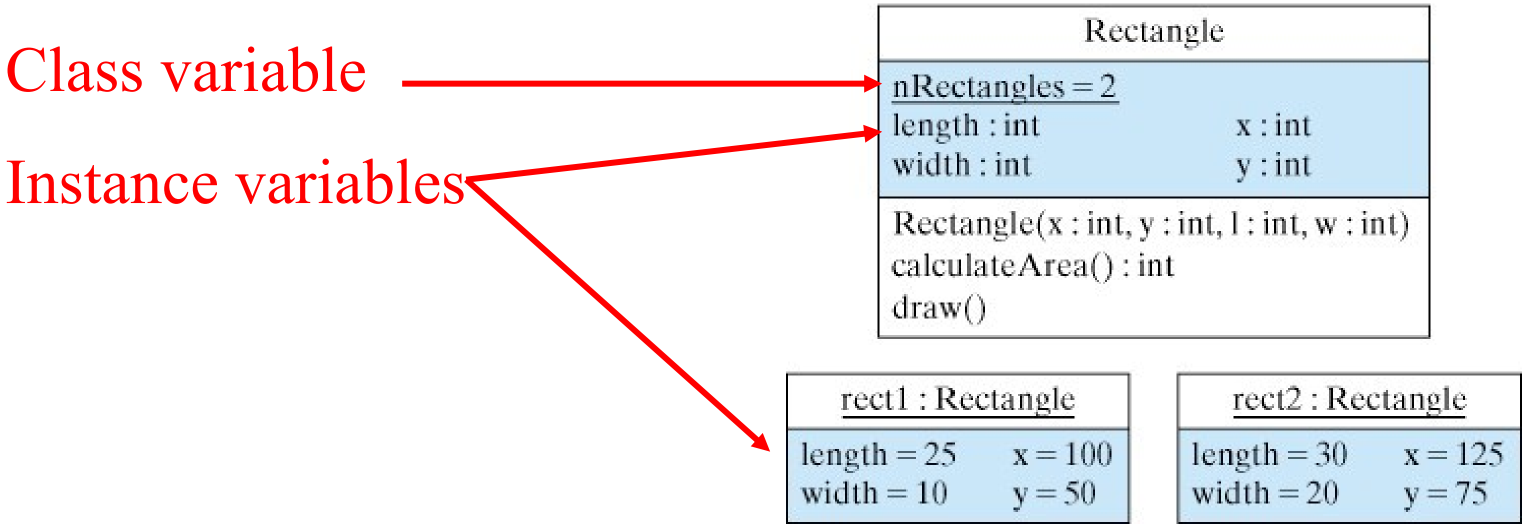


Figure 0.10. The Rectangle class and two of its instances. Note that class variables are underlined in UML.

Declaring Instance Variables

- Examples:

```
              // Instance variables
private int nSticks = 7;
private int player = 1;
```

- In General

$$\mathit{FieldModifiers}_{opt}\ \mathit{TypeId}\ \mathit{VariableId}\ \mathit{Initializer}_{opt}$$

- Fields or instance variables have *class scope*. Their names can be used anywhere within the class.

Constructing Objects

- A *constructor* is a special method, associated with the class, that is used to create instances of that class.
- Example: We call the Rectangle() constructor to create an instance with length=25, width=10 and located at x=100 and y=50.

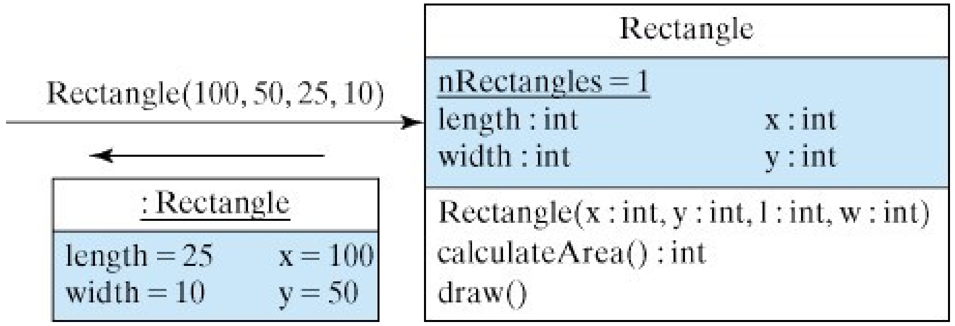


Figure 0.11. Constructing a Rectangle instance.

Class Hierarchy

- Class Hierarchy: Classes can be arranged in a hierarchy from most general to most specific. A *subclass* is more specific than its *superclass*.

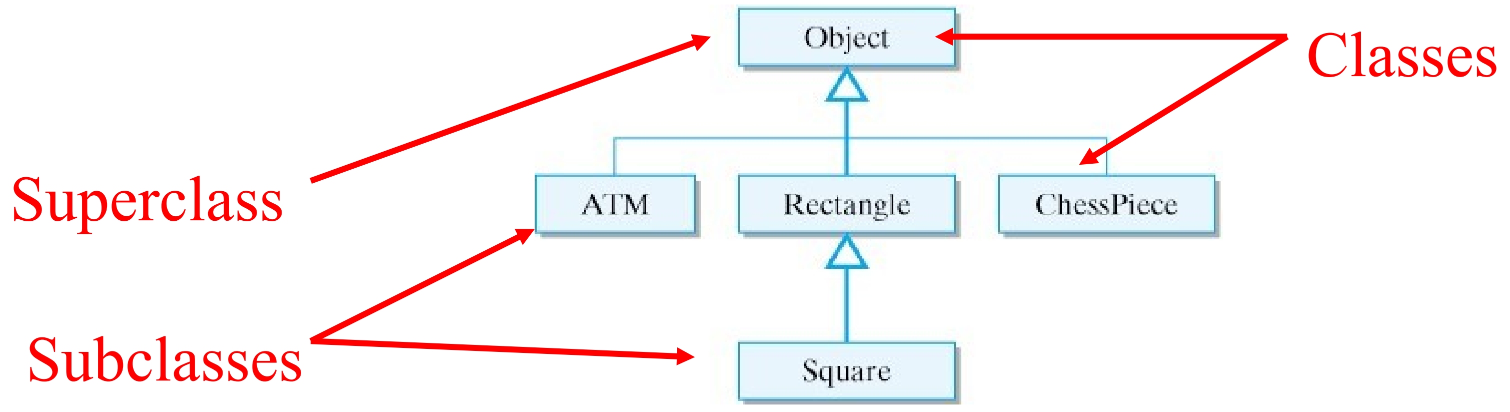


Figure 0.12. A hierarchy of Java classes.

Class Inheritance

- *Class Inheritance*: A subclass inherits elements (variables and methods) from its superclasses.
- Inherited elements can be specialized in the subclass.

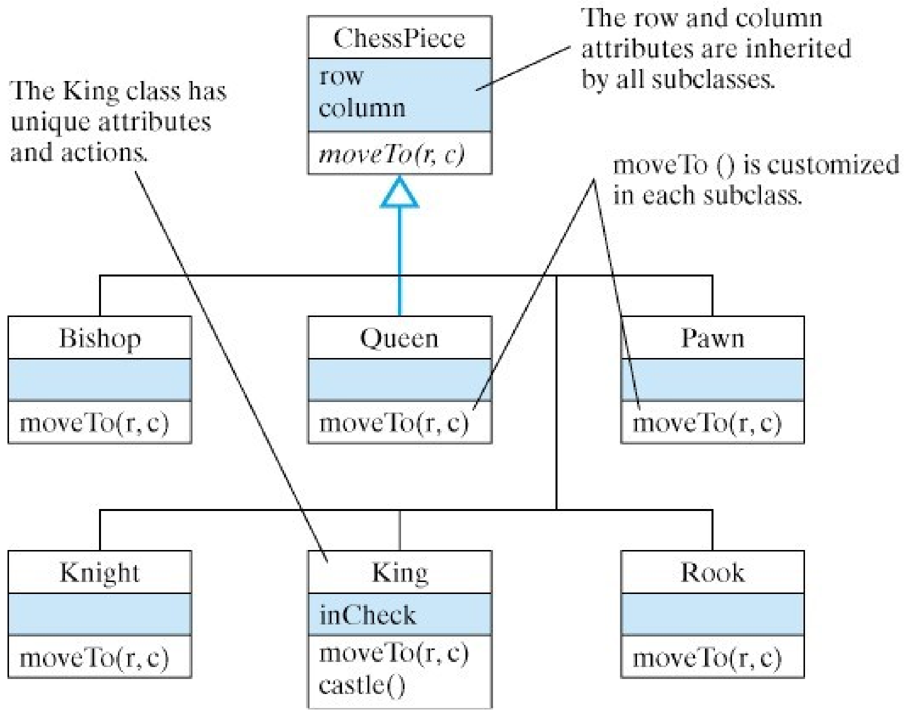

The King class has unique attributes and actions.
ChessPiece
row
column
moveTo(r, c)
The row and column attributes are inherited by all subclasses.
moveTo () is customized in each subclass.
Bishop
moveTo(r, c)
Queen
moveTo(r, c)
Pawn
moveTo(r, c)
Knight
moveTo(r, c)
King
inCheck
moveTo(r, c)
castle()
Rook
moveTo(r, c)

Figure 0.13. The ChessPiece hierarchy.

Principles of Object Orientation

- *Divide-and-Conquer* Principle
  - Problem solving: Break problems (programs) into small, manageable tasks.
  - Example: Sales agent, shipping clerk.
- *Encapsulation* Principle
  - Problem solving: Each object knows how to solve its task and has the information it needs.
  - Example: Sales agent is the sales expert. The shipping clerk is the shipping expert.

Principles of Object Orientation (cont)

- *Interface* Principle
  - Each object has a well-defined interface, so you know how to interact with it.
  - Example: Digital vs. analog clock interfaces.
  - Example: The sales agent must be told the name and part number of the software.
- *Information Hiding* Principle
  - Objects hide details of their expertise.
  - Example: Customer needn’t know *how* the sales agent records the order.

Principles of Object Orientation (cont)

- *Generality* Principle
  - Objects are designed to solve a *kind* of task rather than a singular task.
  - Example: Sales agents sell all kinds of stuff.
- *Extensibility* Principle
  - An object's expertise can be extended or specialized.
  - Example: One sales agent specializes in software sales and another in hardware sales.

OneRowNim Example

```
public class OneRowNim
{ private int nSticks = 7; // Start with 7 sticks.
  private int player = 1;  //Player 1 plays first.

  public void takeOne()
  { nSticks = nSticks - 1;
    player = 3 - player;
  } // takeOne()

  public void takeTwo()
  { nSticks = nSticks - 2;
    player = 3 - player;
  } // takeTwo()

  public void takeThree()
  { nSticks = nSticks - 3;
    player = 3 - player;
  }  // takeThree()

  public void report()
  { System.out.println("Number of sticks left: " + nSticks);
    System.out.println("Next turn by player " + player);
  } // report()
} // OneRowNim1 class
```

Object Oriented Design

- Encapsulation: The OneRowNim class encapsulates a state and a set of actions.
- Information Hiding: OneRowNim's state is defined by private variables, nSticks and player.
- Interface: OneRowNim's interface is defined in terms of its public methods.
- Generality/Extensibility: We can easily extend OneRowNim's functionality.

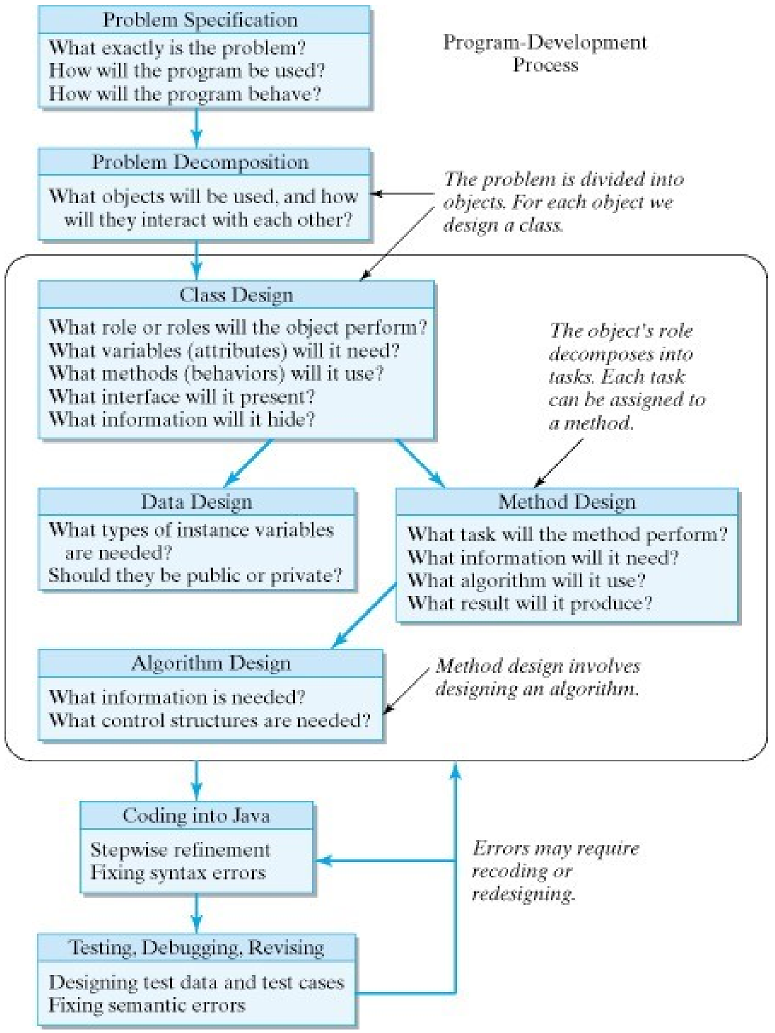
The (Overarching) Abstraction Principle

- *Abstraction* is the ability to focus on the important features of an object when trying to work with large amounts of information.

The Program Development Process

- Problem Specification
- Problem Decomposition
- Design Specification
- Data, Methods, and Algorithms
- Coding into Java
- Testing, Debugging, and Revising

Object-Oriented Program Development


Program-Development Process
Problem Specification
What exactly is the problem?
How will the program be used?
How will the program behave?
Problem Decomposition
What objects will be used, and how will they interact with each other?
The problem is divided into objects. For each object we design a class.
Class Design
What role or roles will the object perform?
What variables (attributes) will it need?
What methods (behaviors) will it use?
What interface will it present?
What information will it hide?
The object's role decomposes into tasks. Each task can be assigned to a method.
Data Design
What types of instance variables are needed?
Should they be public or private?
Method Design
What task will the method perform?
What information will it need?
What algorithm will it use?
What result will it produce?
Algorithm Design
What information is needed?
What control structures are needed?
Method design involves designing an algorithm.
Coding into Java
Stepwise refinement
Fixing syntax errors
Errors may require recoding or redesigning.
Testing, Debugging, Revising
Designing test data and test cases
Fixing semantic errors

Coding into Java

```
public class Riddle extends Object           // Class header
{
    private String question;                 // Instance variables
    private String answer;

    public Riddle(String q, String a) // Constructor method
    {
        question = q;
        answer = a;
    }

    public String getQuestion()        // Access method
    {
        return question;
    } // getQuestion()

    public String getAnswer()          // Access method
    {
        return answer;
    } // getAnswer()

} // Riddle class
```

Syntax

- The ***syntax*** of a programming language is the set of rules that determines whether its statements are correctly formulated.
- Example Rule: All Java statements must end with a semincolon.
- ***Syntax error***: question = q
- Syntax errors can be detected by the compiler, which will report an error message.

Semantics

- The ***semantics*** of a programming language is the set of rules that determine the meaning of its statements.
- Example Rule: In $a + b$, the + operator will add a and b.
- ***Semantic error:*** User intended to add a and b but coded $a - b$.
- Semantic errors cannot be detected by the compiler, because it can’t read the programmer’s mind.

Java Language Elements: Class Definition

```
public class HelloWorld extends Object  // Class header
{                                    // Start of class body
    private String greeting = "Hello World!"; //    Instance variable

    public void greet()                 //    Method definition header
    {                                   //    Start of method body
        System.out.println(greeting);   //    Output statement
    } // greet()                        //    End of greet method body

    public static void main(String args[]) // Method definition header
    {                                      // Start of method body
        HelloWorld helloworld;           //    Reference variable
        helloworld = new HelloWorld();   //    Object instantiation stmt
        helloworld.greet();              //     Method call
    } // main()                          // End of method body
} // HelloWorld                          // End of class body
```

Java Language Elements: Identifiers

- An ***identifier*** must begin with a letter (A to Z, a to z) and may be followed by any number of letters or digits (0 to 9) or underscores (_). An identifier may not be identical to a Java keyword.
- Style: Class names begin with a capital letter and use capitals to distinguish words within the name: ***HelloWord, TextField***
- Style: Variable and method names begin with a lowercase letter and use capitals to distinguish words within the name: ***main(), getQuestion()***

Identifiers

- An *identifier* is a name for a variable, method, or class.
- **Rule**: An identifier in Java must begin with a letter, and may consist of any number of letters, digits, and underscore (_) characters.
- Legal: OneRowNim, takeOne, nSticks
- Illegal: One Row Nim, 5sticks, game$, n!

Data Types

- Java data are classified according to *data type*.
- Types of Objects (Riddle, Rectangle, String) versus *primitive data types* (int, double, boolean).
- A *literal value* is an actual value ("Hello", 5, true).

Type	Keyword	Size in Bits	Examples
boolean	boolean	-	true or false
character	char	16	'A', '5', '+'
byte	byte	8	-128 to +127
integer	short	16	-32768 to +32767
integer	int	32	-2147483648 to +2147483647
integer	long	64	really big numbers
real number	float	32	21.3, -0.45, 1.67e28
real number	double	64	21.3, -0.45, 1.67e28

Java's Primitive Data Types.

Variables

- A *variable* is a typed container that can store a value of a certain type.

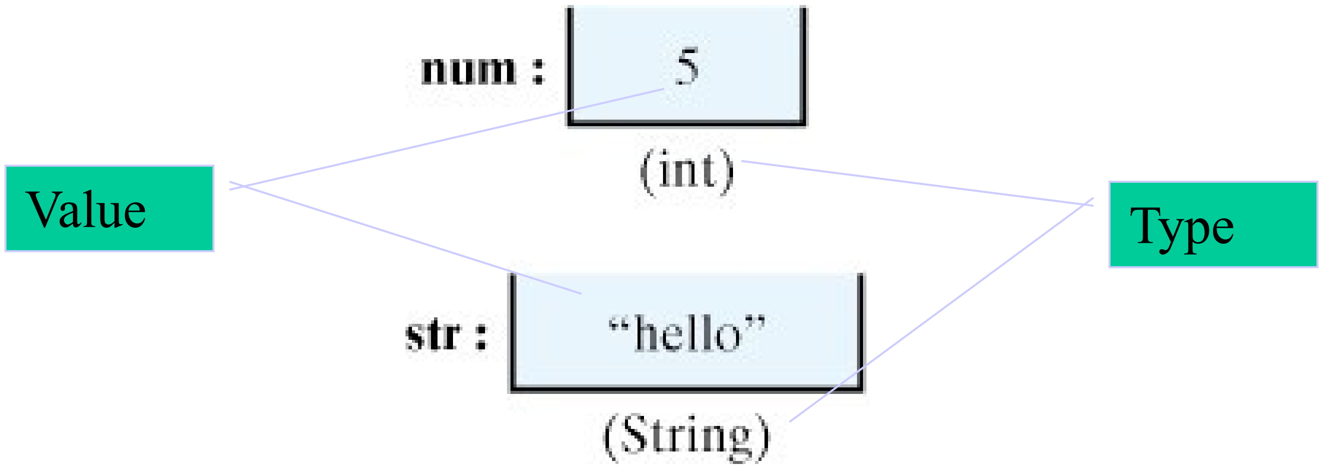


Figure 1.6. A variable is a *typed* container.

Statements

- A *statement* is a segment of code that takes some action in a program.
- A *declaration statement* declares a variable and has the general form:

  Type VariableName ;

```
// Type variableName ;
HelloWorld helloworld ;
int int1, int2;
```

Assignment Statements

- An *assignment statement* stores a value in a variable and has the general form:

  VariableName = Value ;

```
greeting = "Hello World";
num1 = 50;
num2 = 10 + 15;
num1 = num2;
```

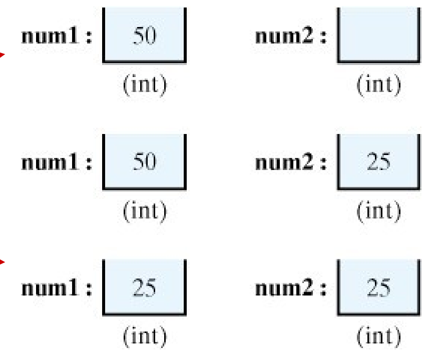


Figure 1.7

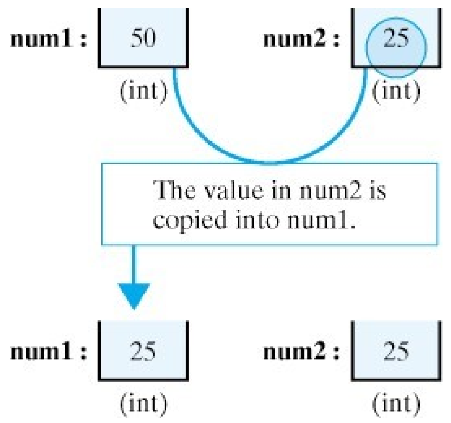


Figure 1.8. In the assignment
num1 = num2 ;
num2's value is copied into num1.

Expressions and Operators

- An *expression* is Java code that specifies or produces a value in the program.
- Expressions use *operators* (=, +, <, …)

```
num1 + num2   // An addition of type int
num1 < num2   // A comparison of type boolean
num = 7       // An assignment expression of type int
square(7)     // A method call expression of type int
num == 7      // An equality expression of type boolean
```

- Expressions occur within statements:

```
num = square(7) ;   // An assignment statement
```

Declaring an Instance Variable

- In general an instance variable declaration takes the following form:

  Modifiers$_{opt}$ *Type VariableName InitialzerExpression*$_{opt}$

```
private String greeting = "Hello World";
int num;
double realNum = 5.05;
static int count = 0;
```

- In these examples the types are String, int, double and the variable names are greeting, num, realNum, and count.

Declaring an Instance Method

- A method definition consists of a *method header*:

  Modifiers$_{opt}$ ReturnType MethodName (ParameterList$_{opt}$)

- Followed by a *method body*, which is *executable* code contained within curly brackets: {…}

```
public void greet()    //   Method definition header
{                      //   Start of method body
    System.out.println(greeting); //   Executable statement
} // greet()           //   End of greet method body
```

```
public static void main(String args[]) // Method definition header
{   HelloWorld helloworld;           //   Reference variable
    helloworld = new HelloWorld();   //   Object instantiation stmt
    helloworld.greet();              //   Method call
} // main()
```

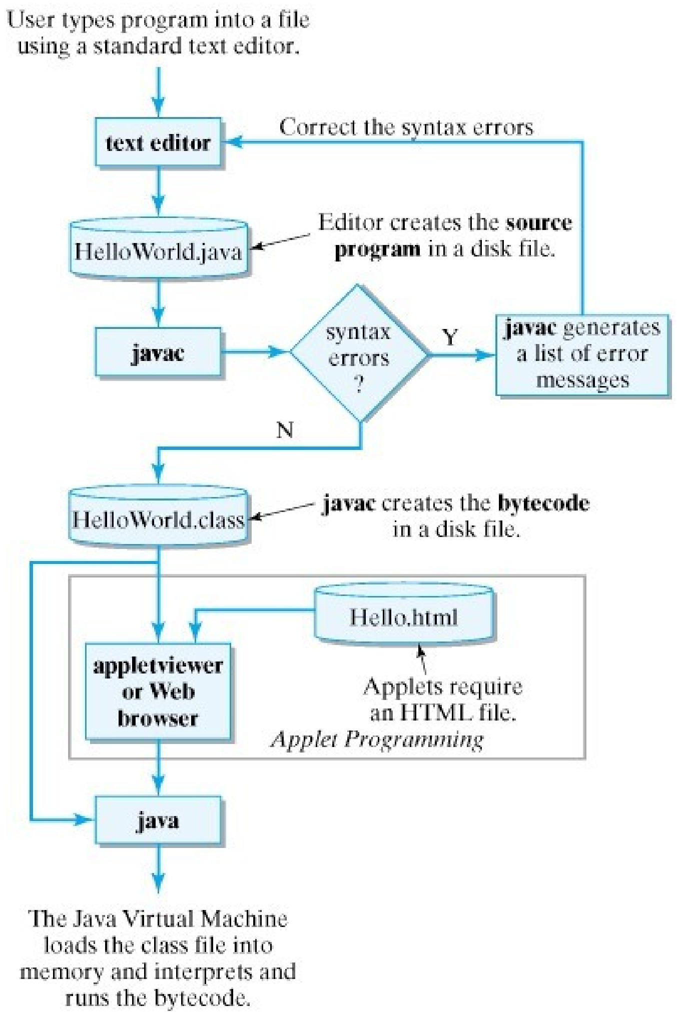
The Java Development Process

- Step 1: Editing the Program
  - Software: Any text editor will do.
- Step 2: Compiling the Program
  - Software: Java Development Kit (JDK)
  - JDK: `javac HelloWorld.java`
- Step 3: Running the Program
  - JDK: `java HelloWorld` (Application)
  - JDK: `appletviewer file.html` (Applet)

Compiling a Java Program

- Compilation translates the *source program* into Java *bytecode.*
  - Bytecode is *platform-independent*
- JDK Command: `javac HelloWorld.java`
- Successful compilation will create the bytecode class file: `HelloWorld.class`

Editing, Compiling, and Running

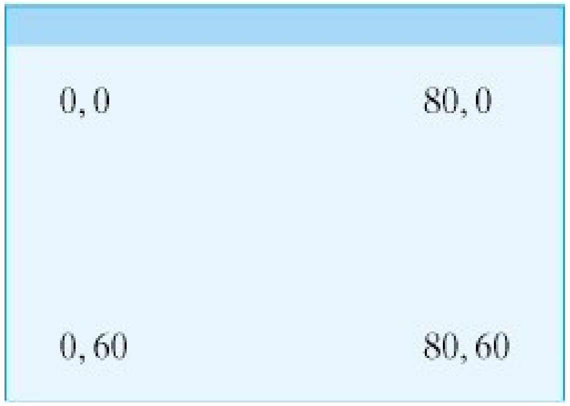

User types program into a file using a standard text editor.
text editor
Correct the syntax errors
HelloWorld.java
Editor creates the **source program** in a disk file.
javac
syntax errors ?
Y
javac generates a list of error messages
N
HelloWorld.class
javac creates the **bytecode** in a disk file.
Hello.html
appletviewer or Web browser
Applets require an HTML file.
Applet Programming
java
The Java Virtual Machine loads the class file into memory and interprets and runs the bytecode.

Using a Graphics Object

- A Graphics object was used in HelloWorldApplet to draw “HelloWorld” on a browser:

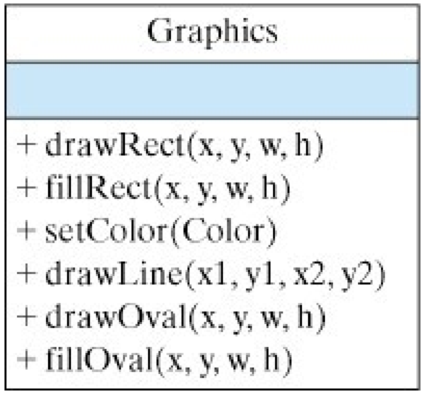
```
public void paint (Graphics g)
{   g.drawString(“Hello World”, 10, 10);
    g.drawString(“Welcome to Java”, 10, 35);
}
```

- In a Java window, the origin of the coordinate system, the point (0,0), is at the top-left corner.



Graphics Drawing Methods

- The **Graphics** class contains useful drawing methods.



- Some Examples:

```
g.setColor(Color.blue);      // Sets the drawing color to blue
g.fillRect(25, 25, 140, 40); // Draws a 140x40 blue rectangle
                             //   at coordinate (25,25)
g.setColor(Color.black);     // Sets the drawing color to black
g.drawRect(25,25,140,40);    // Draws the rectangle outline
```

Class Definition

```java
public class Riddle
{ private String question; //Instance variables
  private String answer;

  public Riddle(String q, String a) // Constructor
  { question = q;
    answer = a;
  } // Riddle constructor

  public String getQuestion() // Instance method
  { return question;
  } // getQuestion()
  public String getAnswer() // Instance method
  { return answer;
  } //getAnswer()
} //Riddle class
```

A public class is accessible to other classes

Instance variables are usually private

An object's public methods make up its *interface*

Public/Private Access

- Instance variables should usually be declared private. This makes them inaccessible to other objects.
- Generally, public methods are used to provide carefully controlled access to the private variables.
- An object's public methods make up its *interface* -- that part of its makeup that is accessible to other objects.

Public/Private Access (cont)

- Possible Error: Public instance variables can lead to an inconsistent state
- Example: Suppose we make nSticks and player public variables.

```
    nim.nSticks = -1;   // Inconsistent
    nim.player = 0;     // State
```

- The proper way to change the game's state:

```
    nim.takeOne();   // takeOne() is public method
```

Method Definition

- Example

```
public void MethodName() // Method Header
{                        // Start of method body
}                        // End of method body
```

- The Method Header

MethodModifiers$_{opt}$ *ResultType MethodName* (*ParameterList*)

↓	↓	↓	↓
public static	void	main	(String argv[])
public	void	takeOne	()
public	void	takeTwo	()
public	void	report	()

Method Definition

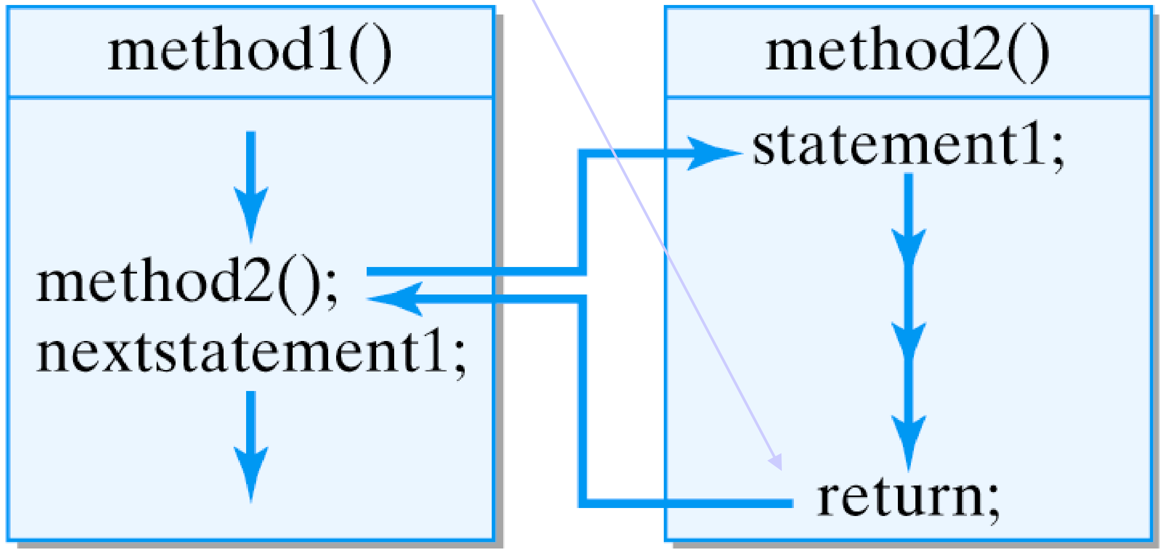
```
public void takeOne()
{
    nSticks = nSticks - 1;
    player = 3 - player;
} // takeOne()
```

Header: This method, named *takeOne*, is accessible to other objects *(public)*, and does not return a value *(void)*.

Body: a block of statements that removes one stick and changes the player's turn

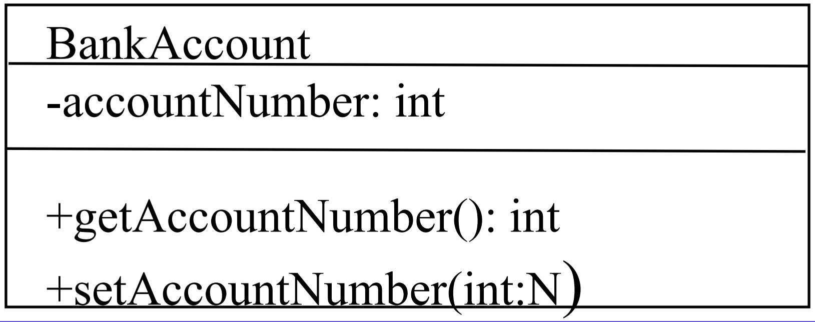
Method Call and Return

- A *method call* causes a program to transfer control to the first statement in the called method.
- A *return statement* returns control to the calling statement.



Types of Methods

- A *accessor method* is a public method used to *get* the value of an object's instance variable.
  - bankAccount.getAccountNumber()
- A *mutator method* is a public method used to *set* the value of an instance variable.
  - bankAccount.setAccountNumber(1098)



Arguments and Parameters

- *Arguments* refer to the values that are used in the *method call* or *method invocation.*

```
game.takeSticks(3);
```

- *Qualified names* (dot notation), are used to refer to methods within other classes.
- The arguments used in the method call must match the *parameters* defined in *method definition.*

```
public void takeSticks(int num) {…}
```

Overloading and Method Signatures

- A method name is *overloaded* if there is more than one method with the same name:

```
public OneRowNim () { }        // Constructor #1
public OneRowNim (int sticks) // Constructor #2
{
   nSticks = sticks;
}
```

- A method is uniquely identified by its method signature, which includes the method name plus the number, order, and types of its parameters.

Passing a Value vs. Passing a Reference

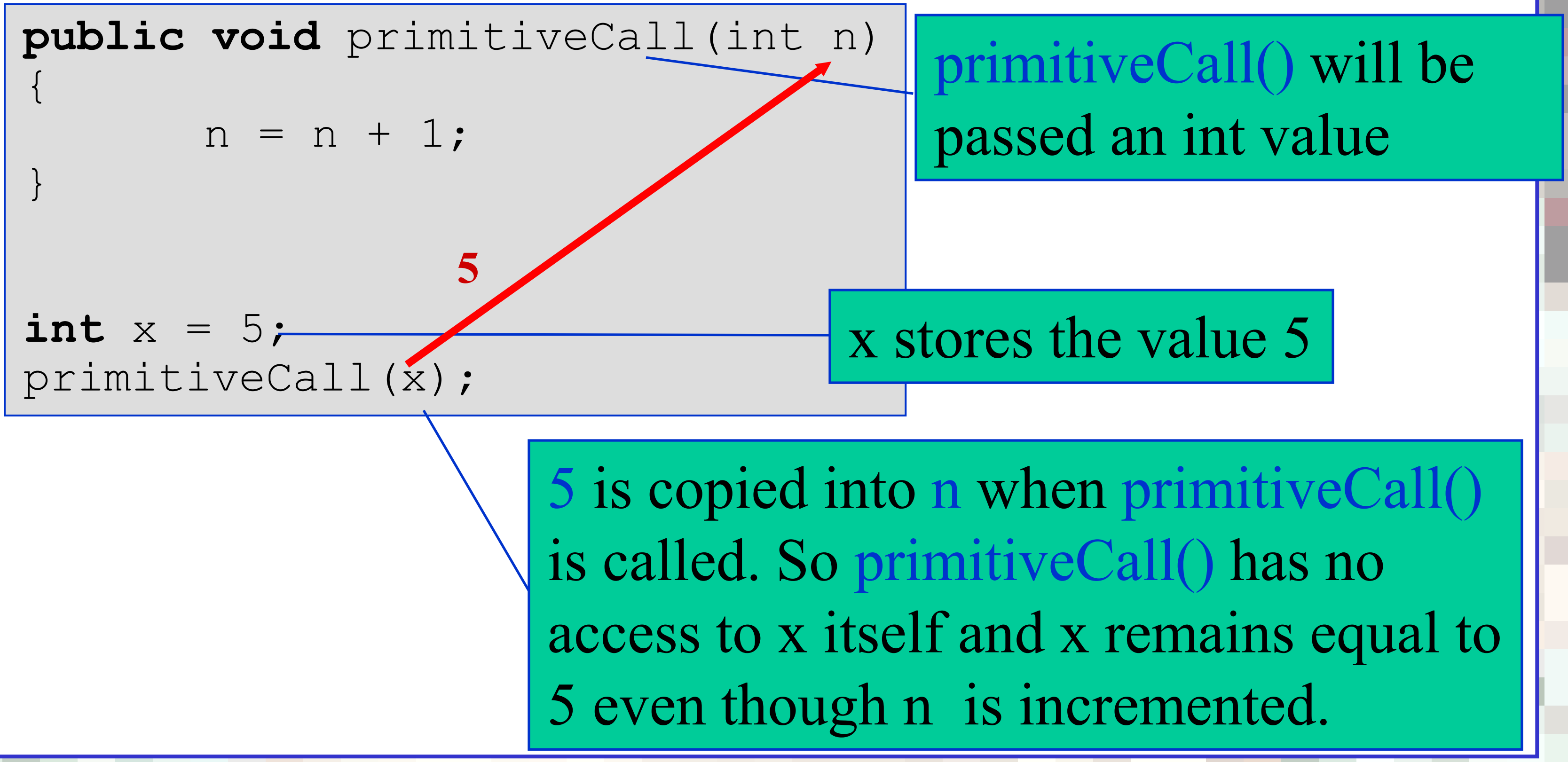
- Question: Does anyone remember the difference between passing a value and passing a reference?

Passing a Value vs. Passing a Reference

- Passing a *primitive value* differs from passing a *reference value*
- Values of type int, boolean, float, and double are examples of *primitive types*. A primitive argument cannot be changed by a method call.
- All objects (String, OneRowNim) are *reference types*. Reference arguments can be changed by a method call.

Passing a Primitive Value

- For primitive type parameters, a ***copy*** of the argument's value is passed to the method.



```
public void primitiveCall(int n)
{
        n = n + 1;
}


int x = 5;
primitiveCall(x);
```

primitiveCall() will be passed an int value

x stores the value 5

5 is copied into n when primitiveCall() is called. So primitiveCall() has no access to x itself and x remains equal to 5 even though n is incremented.

Passing a Reference Value

- For reference parameters, a reference to an object is passed to the method.

```
public void referenceCall(OneRowNim g)
{
    g.takeSticks(3);
}



OneRowNim x = new OneRowNim(7);
referenceCall(x);
```

referenceCall() will be passed a reference to a OneRowNim

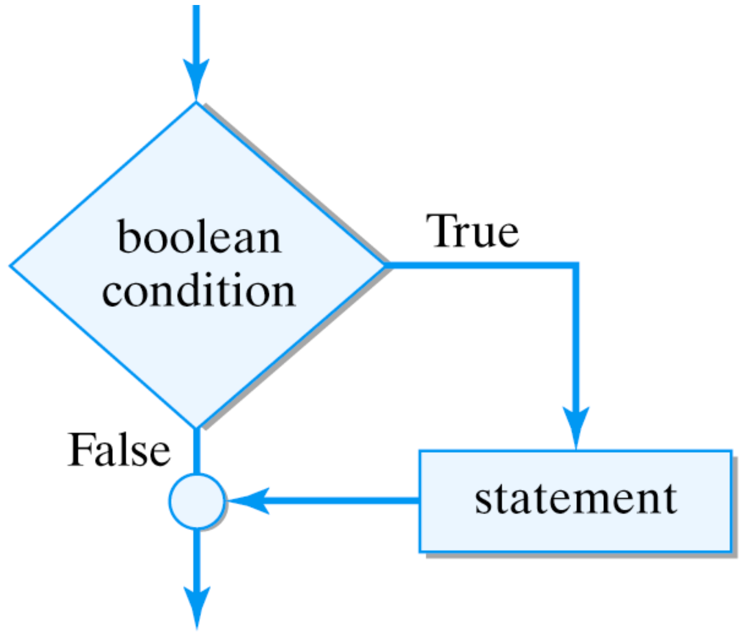
Before: x refers to a OneRowNim with 7 sticks

After: Passing x is like passing the object itself. x's nSticks will be 4 after the method call.

Flow of Control: Control Structures

- ***Selection Control Structures*** allow the program to select one of multiple paths of execution.
- The path is selected based on some conditional criteria, as is the case in a flowchart.

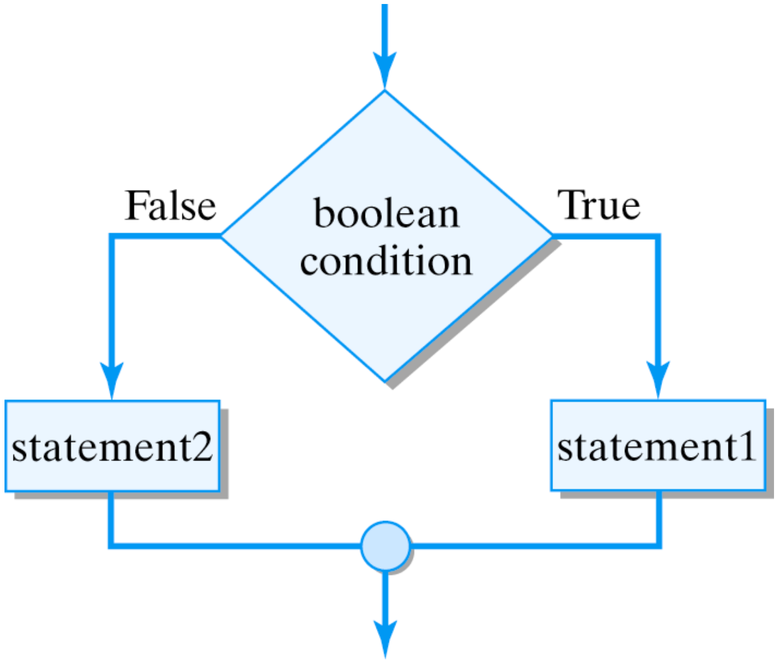
The Simple If Statement



```
if ( boolean expression )
      statement ;
```

- If the *boolean expression* evaluates to true, the statement will be executed. Otherwise, it will be skipped.

The If-Then-Else Statement



```
if ( boolean expression )
      statement1 ;
else
      statement2 ;
```

- If the *boolean expression* is true, execute *statement1*, otherwise execute *statement2*.

Counting Loops

- The *for statement* is used for counting loops.

```
for (int k = 0; k < 100; k++)            // For 100 times
       System.out.println("Hello");      //  Print "Hello"
```

- *Zero-indexing*: the *loop counter* or *loop variable* *k*, *iterates* between 0 and 99.
- For statement syntax:

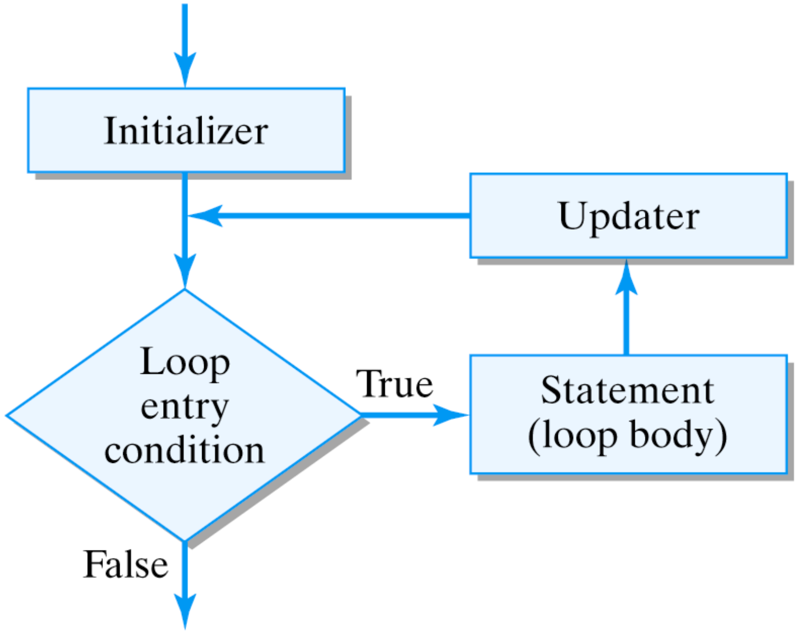
```
for ( initializer ; loop entry condition ; updater )
    for loop body ;
```

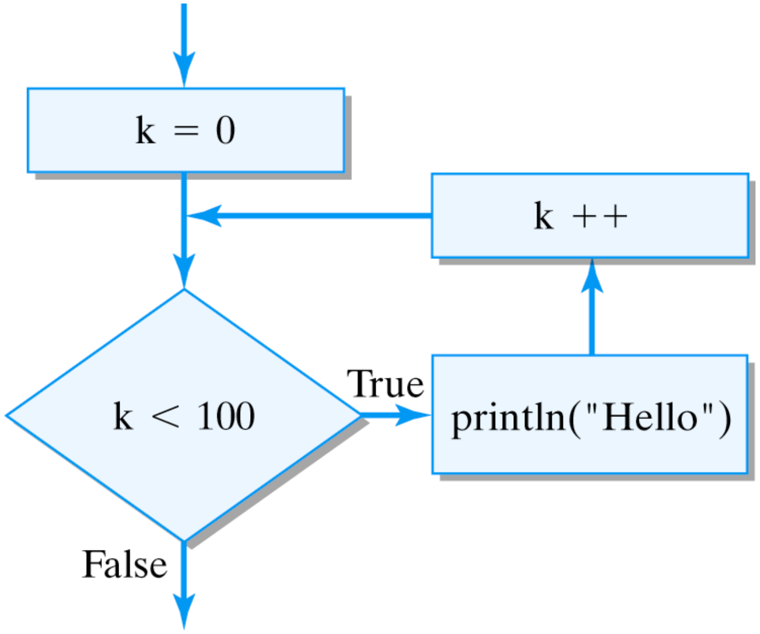
The For Structure

- Syntax:

```
for ( k = 0 ; k < 100 ; k++ )
     System.out.println("Hello");
```

- Semantics:





Nested Loops

- Suppose you wanted to print the following table:

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
4   8   12  16  20  24  28  32  36
```

- You could use a nested for loop. The *outer loop* prints the four rows and in each row, the *inner loop* prints the 9 columns.

```
for (int row = 1; row <= 4; row++) {          // For each of 4 rows
    for (int col = 1; col <= 9; col++)            // For each of 9 columns
        System.out.print(col * row + "\t");        // Print 36 numbers
    System.out.println();                         // Start a new row
} // for row
```

The While Structure

- While structure to sum the squares of the numbers from 1..max:

```
public int sumSquares(int max)
{    int num = 1;
     int sum = 0;
     while (num <= max)
     {    sum = sum + num * num;   // Add square to sum
          num = num + 1;           // Increment num by 1
     } // while
     return sum;
}// sumSquare()
```

Initializers

Loop entry condition: enter the loop if this condition is true

Loop body

Updater: brings num closer to max

- Java's while statement:

while (loop entry condition)
  loop body ;

The while statement has no built-in initializer and updater.

The Do-While Structure

- Problem: How many days will it take for half the lawn to disappear if it loses 2% of its grass a day?

```
public int losingGrass(double perCentGrass) {
    double amtGrass = 100.0;                  // Initialize amount of grass
    int nDays = 0;                            // Initialize day counter
    do {                                      // Repeat
        amtGrass -= amtGrass * LOSSRATE;       //   Update grass
        ++nDays;                               // Increment days
    } while (amtGrass > perCentGrass);         // While 50% grass
    return nDays / 7;                          // Return number of weeks
} // losingGrass()
```

Initializer

Updater

Loop body

Limit bound: Terminate when a limit is reached.

- Java's *do-while statement* :

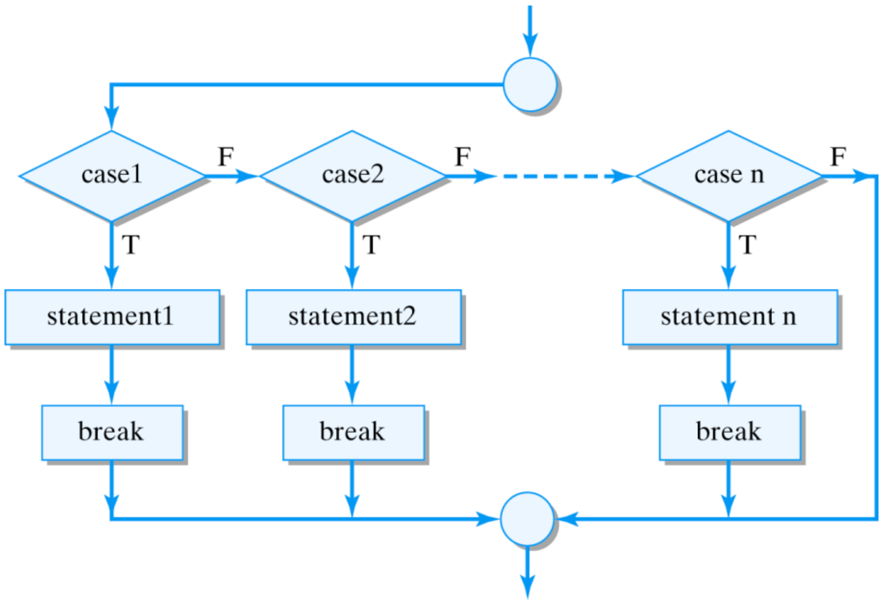
```
do {
     loop body
} while ( loop entry condition )  ;
```

No built-in initializer or updater.

The Switch/Break Structure

- Multiway selection can also be done with the switch/break structure.

```
switch ( integralExpression )
{
    case integralValue2 :
        statement1;
        break;
    case integralValue2 :
        statement2;
        break;
  …
    case integrealValueN :
        statementN;
        break;
    default:
        statementDefault;
}
```

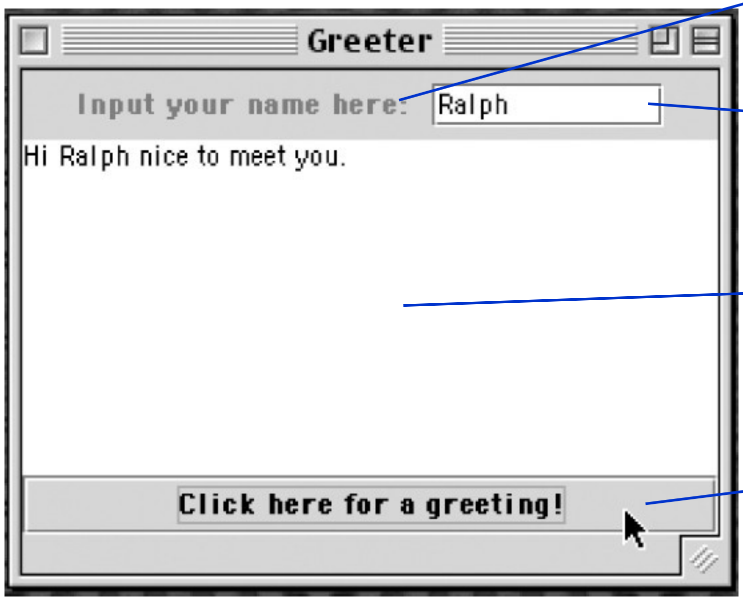


A Graphical User Interface (GUI)

- A GUI uses windows and point-and-click interaction rather than keyboard input.
- A GUI uses *event driven programming* to control the interaction.
- Let's develop a GUI that is *extensible* so that it can be used in a wide variety of applications.

Java's GUI Components

- Java contains many standard GUI components that we can use in our programs.



Hi Ralph nice to meet you.

A JLabel for the prompt.

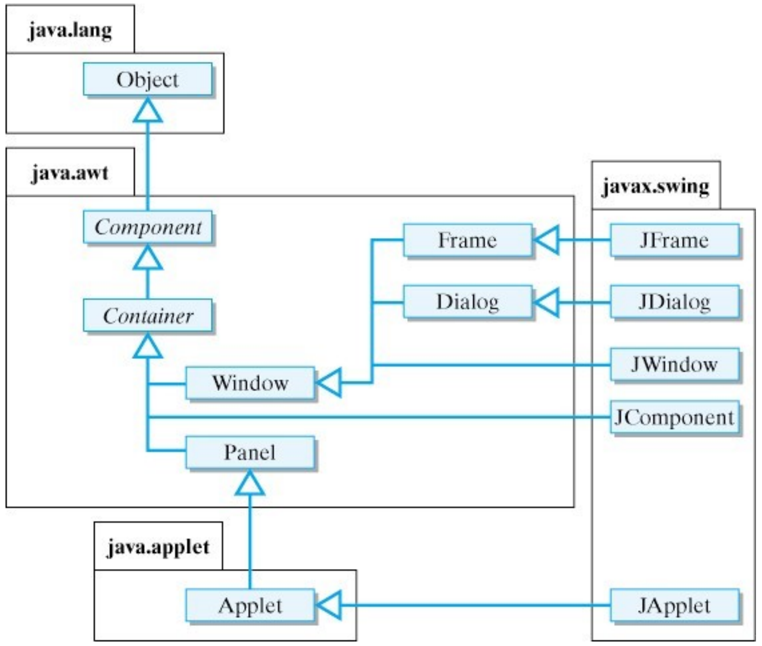
A JTextField for input.

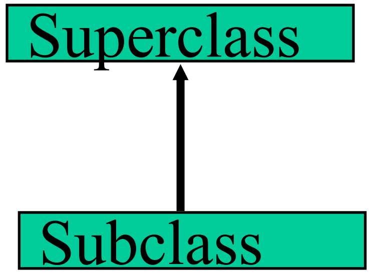
A JTextArea for displaying output.

A JButton for control.

Top-Level Swing and AWT Classes

- The javax.swing and java.awt packages contain Java's GUI components.
- Note the inheritance relationships.

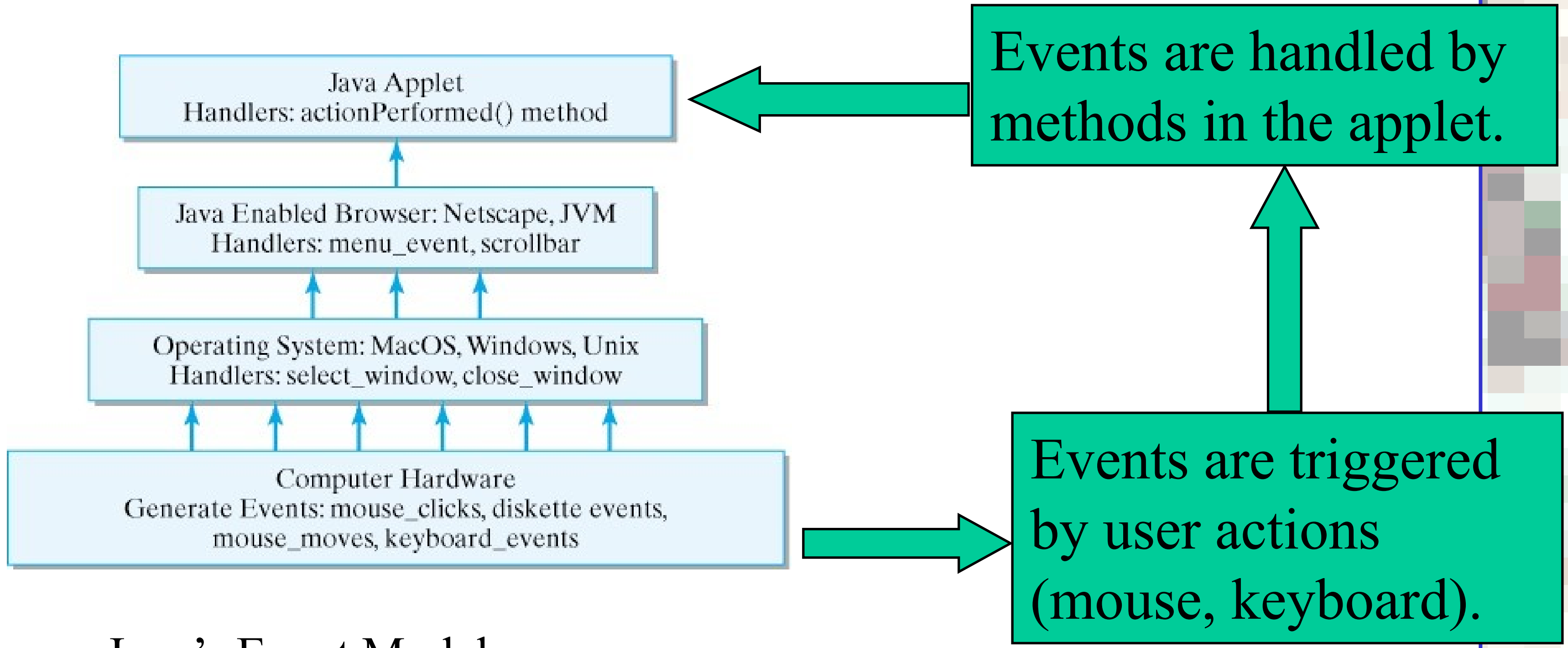




- A subclass inherits characteristics of its superclasses.

Controlling the GUI's Action

- GUIs use *event-driven programming*.


Java Applet
Handlers: actionPerformed() method
Java Enabled Browser: Netscape, JVM
Handlers: menu_event, scrollbar
Operating System: MacOS, Windows, Unix
Handlers: select_window, close_window
Computer Hardware
Generate Events: mouse_clicks, diskette events, mouse_moves, keyboard_events
Events are handled by methods in the applet.
Events are triggered by user actions (mouse, keyboard).

Java's Event Model

The ActionListener Interface

- A *listener* is an object that listens for certain types of events and then takes action.
- An ActionListener listens for ActionEvents, the kind that occur when you click on a JButton or hit return in a JTextField.
- An *interface* is a class that contains abstract methods and constants (no instance variables).
- An *abstract* method is one that lacks an implementation. It has no body.

```
public abstract interface ActionListener extends EventListener
{
    public abstract void actionPerformed(ActionEvent e);
}
```

Implementing an ActionListener

- If we want our program to serve as a listener for action events, we must implement the ActionListener interface.
- This means we must implement the actionPerformed() method.
- By associating the ActionListener interface with a JButton, we cause Java to call actionPerformed() whenever the JButton is clicked.

An ActionListener Example

- GreeterGui.java implements an ActionListener.

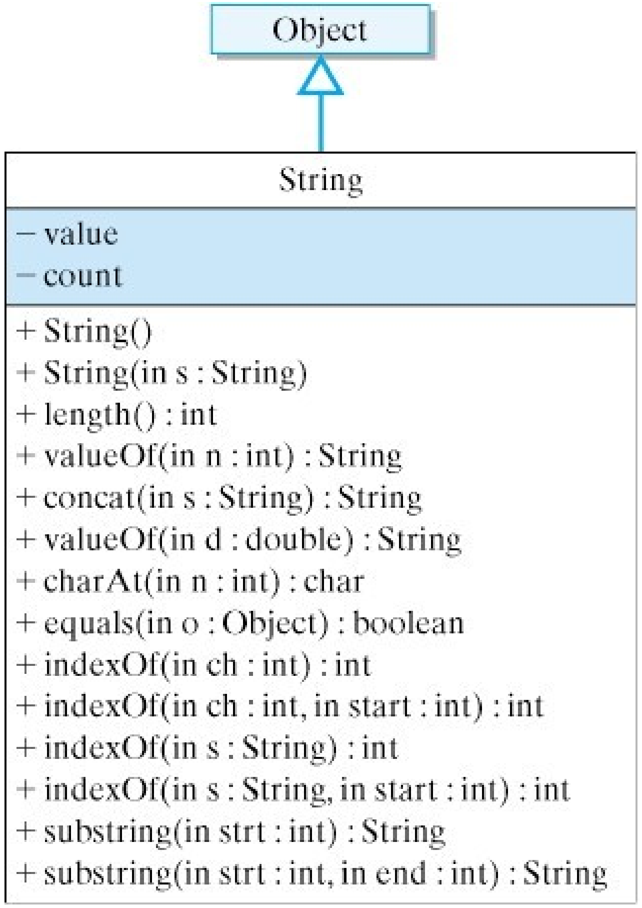
```
public class GreeterGUI extends Frame  implements ActionListener
{   ...
    private JButton goButton;     // Declare the button variable
    ...
    public void buildGUI()
    {  ...
       goButton = new JButton("Click here for a greeting!"); // Create the button
       goButton.addActionListener(this);  // Associate it with this object
       ...
    }
    ...
    public void actionPerformed(ActionEvent e) // This method is called
    {   if (e.getSource() == goButton)         //  whenever goButton is clicked.
        {   String name = inField.getText();
            display.append(greeter.greet(name) + "\n");
        }
    }
    ...
}
```

The keyword this refers to this GreeterGUI object, which is defined as an ActionListener.

String Basics

- A java.lang.String object is a sequence of characters plus a collection of methods for manipulating strings.
- Unlike other Java objects, Strings have certain characteristics in common with the primitive data types.
- For example, strings can have literals. A String *literal* is a sequence of zero or more characters contained in double quotes -- for example, “Socrates” and “” (empty string).

The String Class


Object
String
− value
− count
+ String()
+ String(in s : String)
+ length() : int
+ valueOf(in n : int) : String
+ concat(in s : String) : String
+ valueOf(in d : double) : String
+ charAt(in n : int) : char
+ equals(in o : Object) : boolean
+ indexOf(in ch : int) : int
+ indexOf(in ch : int, in start : int) : int
+ indexOf(in s : String) : int
+ indexOf(in s : String, in start : int) : int
+ substring(in strt : int) : String
+ substring(in strt : int, in end : int) : String

Concatenating Strings

- When surrounded on either side by a String, the + symbol is used as a binary concatenation operator. It has the effect of joining two strings together.

```
String lastName = "Onassis";
String jackie = new String("Jacqueline " + "Kennedy " + lastName);
```

"Jacqueline Kennedy Onassis"

- Primitive types are automatically promoted to strings when mixed with concatenation operators.

```
System.out.println("The square root of 25 = " + 5);
```

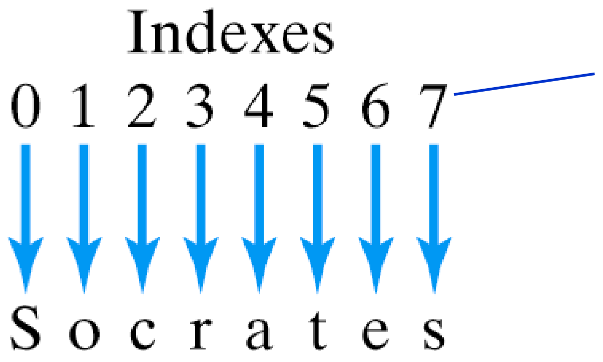
Output: The square root of 25 = 5

Indexing Strings

- The number of characters in a string is its *length*.

```
String string1 = "";                        // string1.length()   ==> 0
String string2 = "Hello";                    // string2.length()   ==> 5
String string3 = "World";                    // string3.length()   ==> 5;
String string4 = string2 + " " + string3;   // string4.length()   ==> 11;
```

- The position of a character within a string is called its *index*. Strings are *zero indexed* -- the first character is at index 0.



Note: Because of *zero indexing*, the last character in a string of 8 characters is at index 7.

Converting Data to String

- The String.valueOf() methods are *class methods* that convert primitive types into String objects.

```
static public String valueOf( primitive type  );
```

```
String number = new String (String.valueOf(128)); // Creates "128"
String truth = new String (String.valueOf(true)); // Creates "true"
String bee = new String (String.valueOf('B'));    // Creates "B"
String pi = new String(String.valueOf(Math.PI));  // Creates "3.14159"
```

Recall that one refers to class methods by using the class name as the qualifier.

Note the difference between 'B' and "B"

String Identity vs. String Equality

- Methods for comparing strings:

```
public boolean equals(Object anObject);  // Overrides Object.equals()
public boolean equalsIgnoreCase(String  anotherString)
public int compareTo(String  anotherString)
```

- Two strings are equal if they have the same letters in the same order:

```
String s1 = "hello";
String s2 = "Hello";
s1.equals(s2)        // false
s1.equals("hello”); // true
```

- Error: Using == to compare two strings. For objects, o1 == o2 means o1 and o2 are *identical*.
- The == operator is equivalent to Object.equals() method: o1.equals(o2) means o1 and o2 are identical.

Array Introduction

- An *array* is a named collection of *contiguous* storage locations holding data of the *same type*.
- Arrays elements: *referenced by position* within a structure rather than by name.
- Example: 26 buttons 'A' to 'Z'.

Without Arrays

```
JButton button1 = new JButton("A");
JButton button2 = new JButton("B");
JButton button3 = new JButton("C");
JButton button4 = new JButton("D");
JButton button5 = new JButton("E");
JButton button6 = new JButton("F");
…
JButton button26 = new JButton("Z");
```

Reference by name

With Arrays

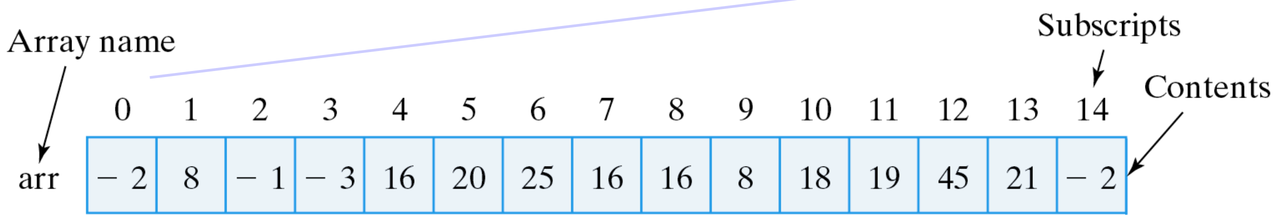
```
JButton letter[] = new JButton[26];
for (int k = 0; k < 26; k++) {
    char ch = (char)('A' + k);
    letter[k] = new JButton(""+ ch);
}
```

The kth JButton in an array.

One-Dimensional Arrays

- An array element is referred to its *position* within the array.
- For an *n-element* array named *arr*, the elements are named *arr[0], arr[1], arr[2], ...,arr[n-1].*
- The following array contains 15 int elements.

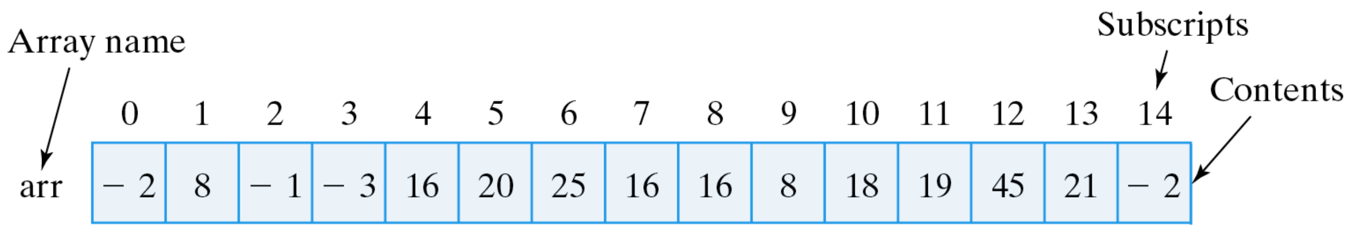
Arrays are *zero indexed*.



- Array syntax : *arrayname* [*subscript*]

  where *arrayname* is the array name and *subscript* is an integer giving the element's relative position.

Referring to Array Elements



- Valid References: Suppose *j* is 5 and *k* is 7.

```
arr[4]         // Refers to 16
arr[j]         // Is arr[5] which refers to 20
arr[j + k]     //  Is arr[5+7] which is arr[12] which refers to 45
arr[k % j]     //  Is arr[7%5] which is arr[2] which refers to -1
```

- Invalid References:

```
arr[5.0]       // 5.0 is a float and can't be an array subscript
arr['5']       // '5' in Unicode would be out of bounds
arr["5"]       // "5" is a string not an integer
arr[-1]        // Arrays cannot have negative subscripts
arr[15]        // The last element of arr has subscript 14
arr[j*k]       // Since j*k equals 35
```

Are Arrays Objects?

- Arrays are (mostly) treated as objects:
  - Instantiated with the new operator.
  - Have instance variables (e.g., length).
  - Array variables are *reference variables*.
  - As a parameter, a reference to the array is passed rather than copies of the array's elements.
- But…
  - Arrays don't fit into the Object hierarchy.
  - Arrays don't inherit properties from Object.

Some Array Terminology

- An *empty array* is contains zero variables.
- The variables are called *components*.
- The *length* of the array is the number of components it has.
- Each component of an array has the same *component type*.
- A *one-dimensional array* has components that are called the array's elements. Their type is the array's *element type*.
- An array's elements may be of any type, including primitive and reference types.

Declaring and Creating an Array

- Creating a one-dimensional array: Indicate both the array's *element type* and its *length*.
- Declare the array's name and create the array itself.

```
int arr[];              // Declare a name for the array
arr = new int[15];      // Create the array itself
```

- Combine two steps into one:

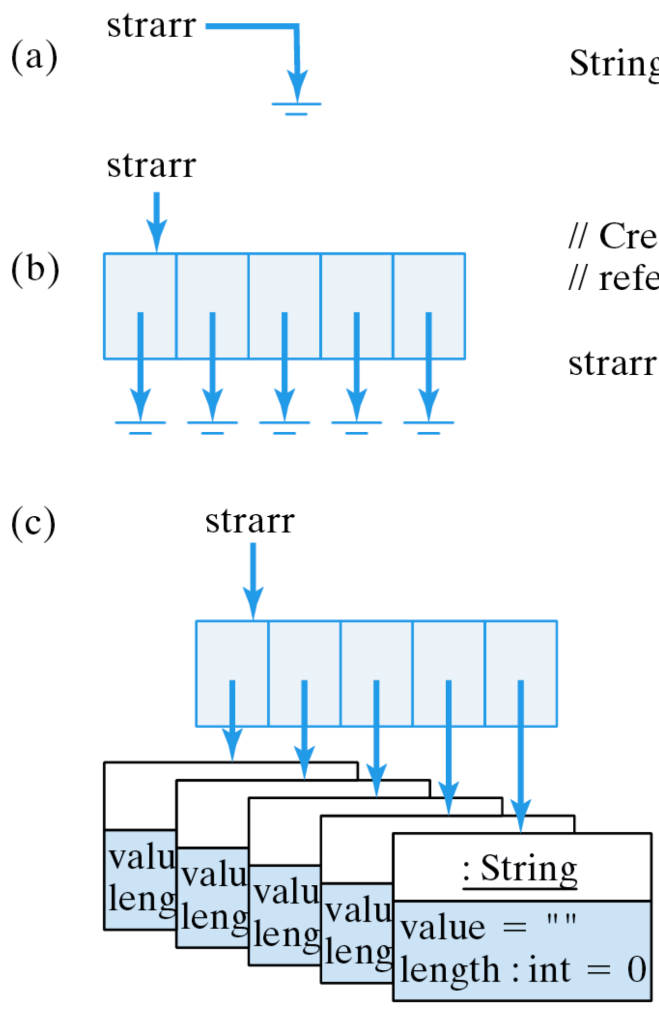
```
int arr[] = new int[15];
```

The array's name is *arr*.

The array contains 15 int variables.

- 15 variables: arr[0], arr[1], .., arr[14] (*zero indexed*)

Creating an Array of Strings



```
String strarr[]; // Null array variable
```

Declare array variable.

```
// Creates an array of null String
// references.

strarr = new String [5];
```

Instantiate the array.

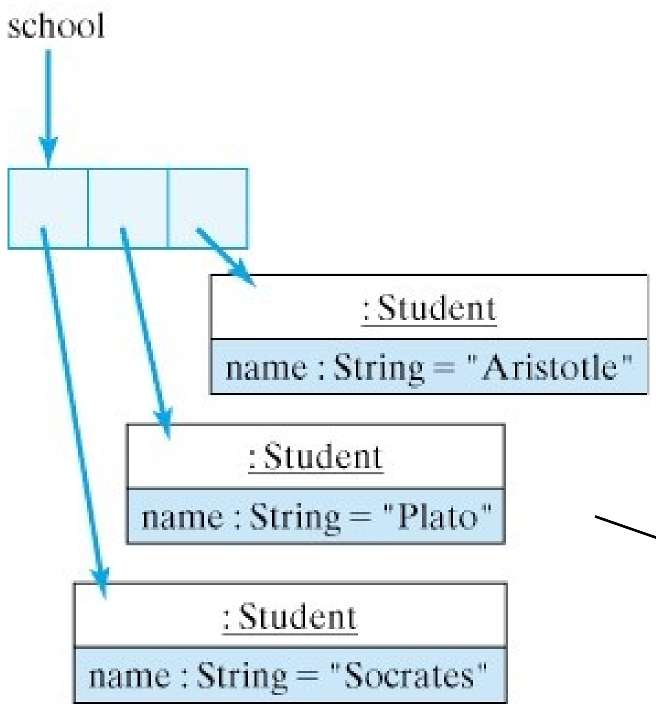
```
// Creates 5 Strings and
/  assigns them to the array

for (int k = 0; k < strarr.length; k++ )
    strarr[k] = new String();
```

Store 5 Strings in it.

Creating an Array of Students

```
Student school[] = new Student[3];     // Create an array of 3 Students
school[0] = new Student("Socrates");    // Create the first Student
school[1] = new Student("Plato");       // Create the second Student
school[2] = new Student("Aristotle");   // Create the third Student
```



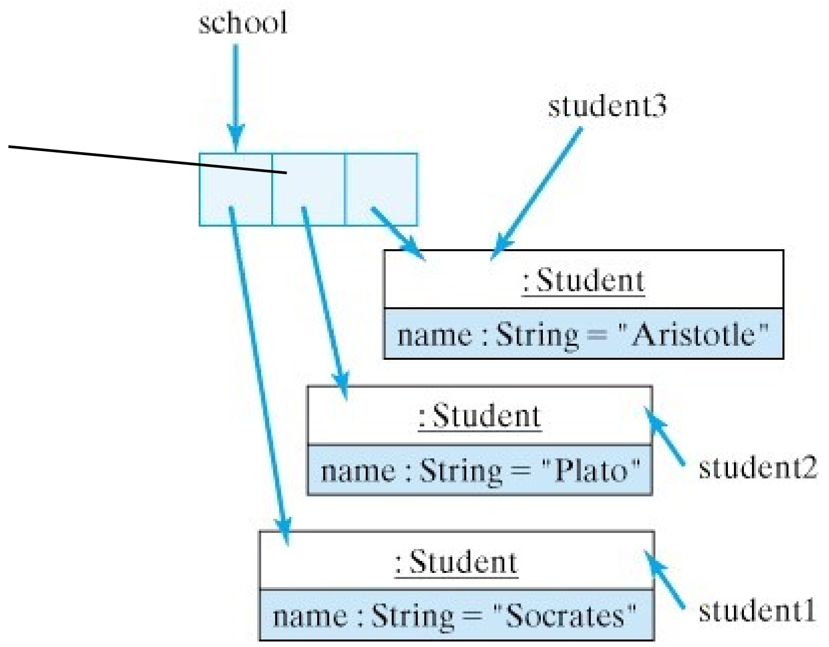
- *Debugging Tip:* Creating a new array does not also create the objects that are stored in the array. They must be instantiated separately.

There are four objects here. One array and 3 Students.

Creating an Array of Students

```
Student student1 = new Student (“Socrates”);
Student student2 = new Student (“Plato”);
Student student3 = new Student (“Aristotle”);
Student school[] = new Student [3];
school[0] = student1;
school[1] = student2;
school[2] = student3;
```

The array stores references to objects, not the objects themselves.



Initializing Arrays

- Array elements are initialized to *default values*:
  - Integer and real types are initialized to *0*.
  - Reference types (objects) are initialized to *null*.
- Arrays can be assigned initial values when they are created:

```
int arr[] = { -2,8,-1,-3,16,20,25,16,16,8,18,19,45,21,-2 } ;

String strings[] = { "hello", "world", "goodbye", "love" } ;
```

- *Java Language Rule*: When an array initialization expression is used, don't use the keyword *new* to create the array.

Assigning and Using Array Values

- Subscripted array variables are used like other variables:

```
arr[0] = 5;
arr[5] = 10;
arr[2] = 3;
strings[0] = "who";
strings[1] = "what";
strings[2] = strings[3] = "where";
```

- A loop to assign the first 15 squares, 1, 4, 9 …, to the array arr:

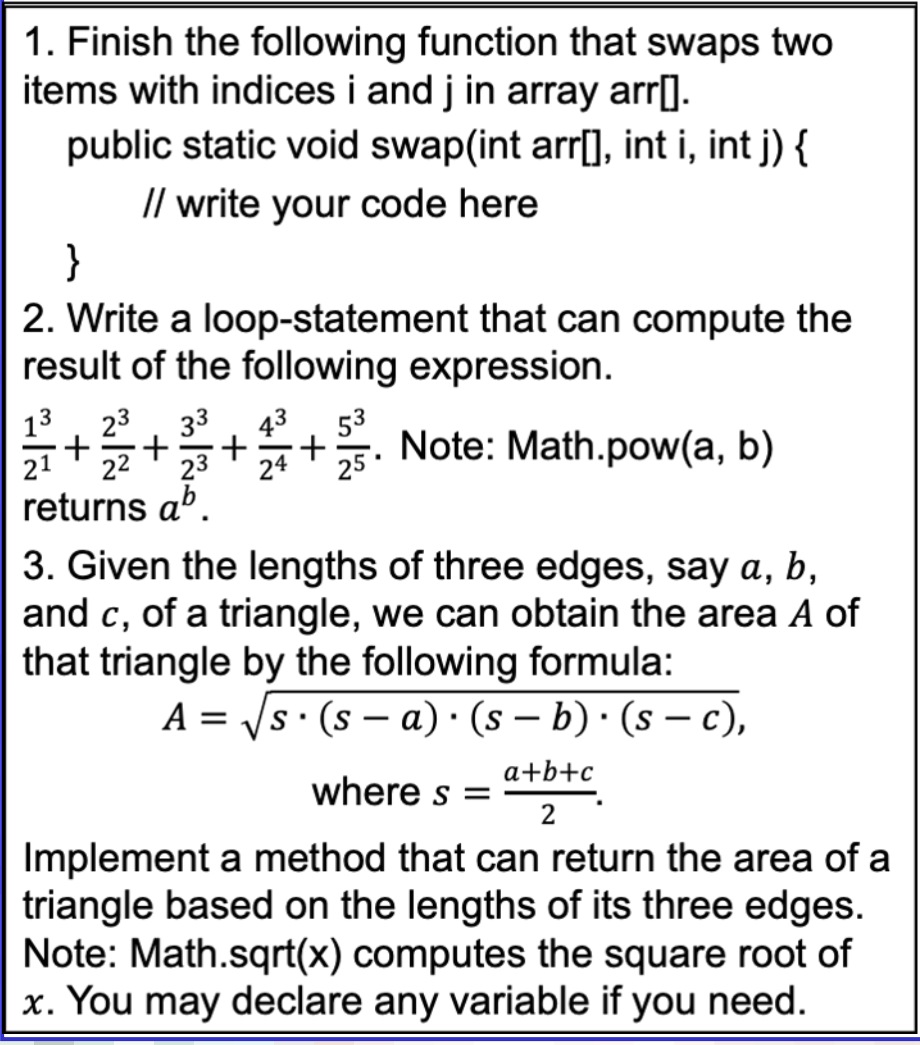
```
for (int k = 0; k < arr.length; k++)
        arr[k] = (k+1) * (k+1);
```

- A loop to print the values of arr:

```
for (int k = 0; k < arr.length; k++)
        System.out.println(arr[k]);
```

Note: *length* is an instance variable, not a method.

Quiz



1. Finish the following function that swaps two items with indices i and j in array arr[].

```
public static void swap(int arr[], int i, int j) {
        // write your code here
}
```

2. Write a loop-statement that can compute the result of the following expression.

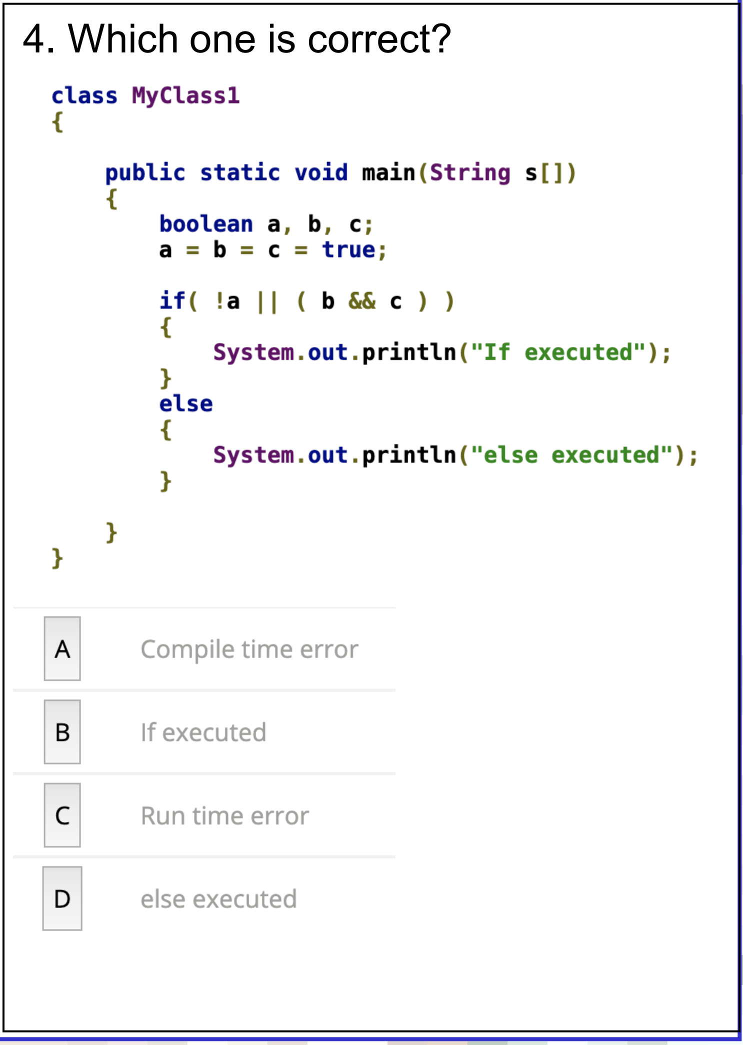
$\frac{1^3}{2^1}+\frac{2^3}{2^2}+\frac{3^3}{2^3}+\frac{4^3}{2^4}+\frac{5^3}{2^5}$. Note: Math.pow(a, b) returns a^b.

3. Given the lengths of three edges, say a, b, and c, of a triangle, we can obtain the area A of that triangle by the following formula:

$$A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)},$$

where $s = \frac{a+b+c}{2}$.

Implement a method that can return the area of a triangle based on the lengths of its three edges. Note: Math.sqrt(x) computes the square root of x. You may declare any variable if you need.



4. Which one is correct?

```
class MyClass1
{

    public static void main(String s[])
    {
        boolean a, b, c;
        a = b = c = true;

        if( !a || ( b && c ) )
        {
            System.out.println("If executed");
        }
        else
        {
            System.out.println("else executed");
        }

    }
}
```

A	Compile time error
B	If executed
C	Run time error
D	else executed

Quiz

1. Finish the following function that swaps two items with indices i and j in array arr[].

```
public static void swap(int arr[], int i, int j) {
          // write your code here
}
```

Solution:

```
public static void swap(int arr[], int i, int j) {
        int t;
        t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
}
```

Quiz

2. Write a loop-statement that can compute the result of the following expression.

$\frac{1^3}{2^1} + \frac{2^3}{2^2} + \frac{3^3}{2^3} + \frac{4^3}{2^4} + \frac{5^3}{2^5}$. Note: Math.pow(a, b) returns a^b.

Solution:

```
double sum=0;
for (int i=1; i<6; i++) {
    sum += Math.pow(i,3)/Math.pow(2, i);
}
```

Quiz

3. Given the lengths of three edges, say a, b, and c, of a triangle, we can obtain the area A of that triangle by the following formula:

$$A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}, \text{ where } s = \frac{a+b+c}{2}.$$

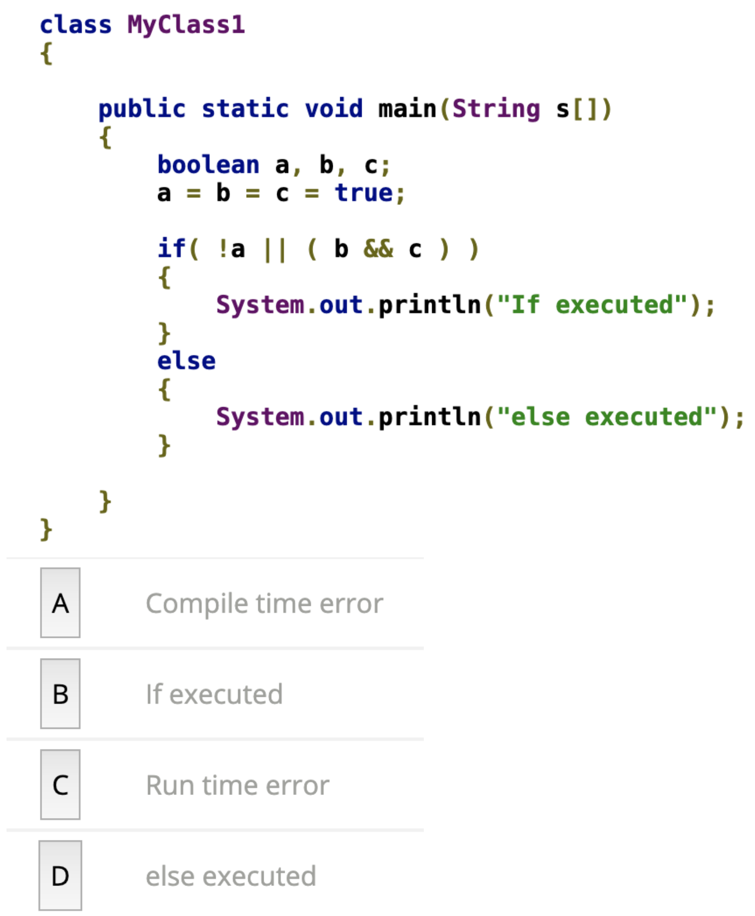
Implement a method that can return the area of a triangle based on the lengths of its three edges. Note: Math.sqrt(x) computes the square root of x. You may declare any variable if you need.

Solution:

```
public double area(double a, double b, double c) {
    double area;
    double s;
    s=(a+b+c)/2;
    area=Math.sqrt(s*(s-a)*(s-b)*(s-c));
    return area;
}
```

Quiz

4. Which one is correct? Answer: B.



```
class MyClass1
{

    public static void main(String s[])
    {
        boolean a, b, c;
        a = b = c = true;

        if( !a || ( b && c ) )
        {
            System.out.println("If executed");
        }
        else
        {
            System.out.println("else executed");
        }

    }
}
```

A Compile time error

B If executed

C Run time error

D else executed

Questions

- Any questions?