Java(TM) SE Development Kit 6 - Complete

Java Notes

Cutajar & Cutajar

START

Objects & Classes

To distinguish between an object and a class one may use the analogy of a car. When we speak of a class we speak of a type of object for example a class of a car, maybe a volkswagen polo.

A VW polo is not a distinct object – it has no actual colour or registration number. On the other hand when we speak of my car of type VW polo we are specifying a distinct object. It's colour is blue and the registration number is GAT 748. This is an object - a specific VW polo car.

Objects

You can have many objects of the same class and each object we can call an instance of that class. Tom Jerry, and Mary can each have a VW polo car; each an instance of a VW polo class.

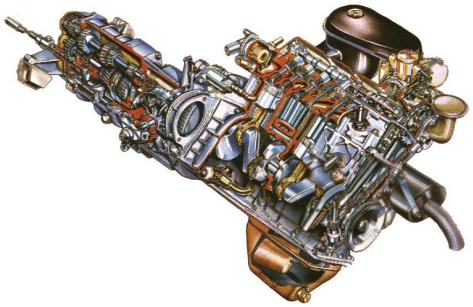
The designer didn't design my car but designed the VW polo of which my car is an instance. So in programming we design a class and then create new instances of that class.

Class

Object Example: **GAT 748 : VW Polo**

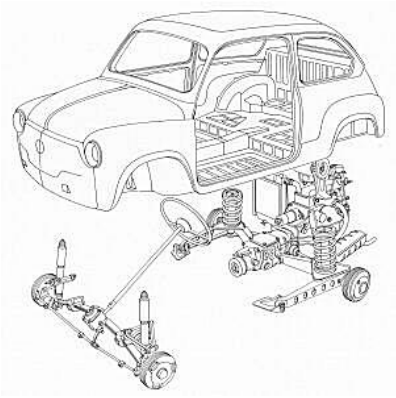
Abstraction

"The essence of abstraction is to extract essential properties while omitting inessential details.". To keep to our analogy, an abstraction can be considered as a car without any details of how it functions. All that I am concerned, is that my car behaves exactly like any her car of its type, and knowing how to use one enables me to drive any car of that type since all I am concerned is that it provide all facilities offered by a normal car.



Information Hiding

Information hiding is the notion that users of a component need to know only the essential details of how to initialize and access the component, and do not need to know the details of the implementation.". The user of the car is not concerned with the inner workings of the engine, but only to how to start the car and the method to drive it.



Encapsulation

"The concept of encapsulation refers to building a capsule, in the case a conceptual barrier, around some collection of things.". All functions of the car are encapsulated within the bonnet. This makes the object much more easy to handle as all same

Methods & Data Values

An object is composed of data values and methods.

Methods are ways to use an object and they normally receive or return a value to or from the object. In our car analogy, a driver drives the car by changing gear, increasing the pressure on the accelerator etc. In order to use a car, a driver does not need to know the inner working of the car, but only the methods provided. So to be able to use the object its user must only know how to use the methods provided by the object.

Data values are values contained within the object. They are the properties of that object. These can be instance data values which belong to only one particular instance, say colour and speed of the car, or a class data value as say the cylinder capacity of that class of cars, which remains the same for all cars belonging to the same class.

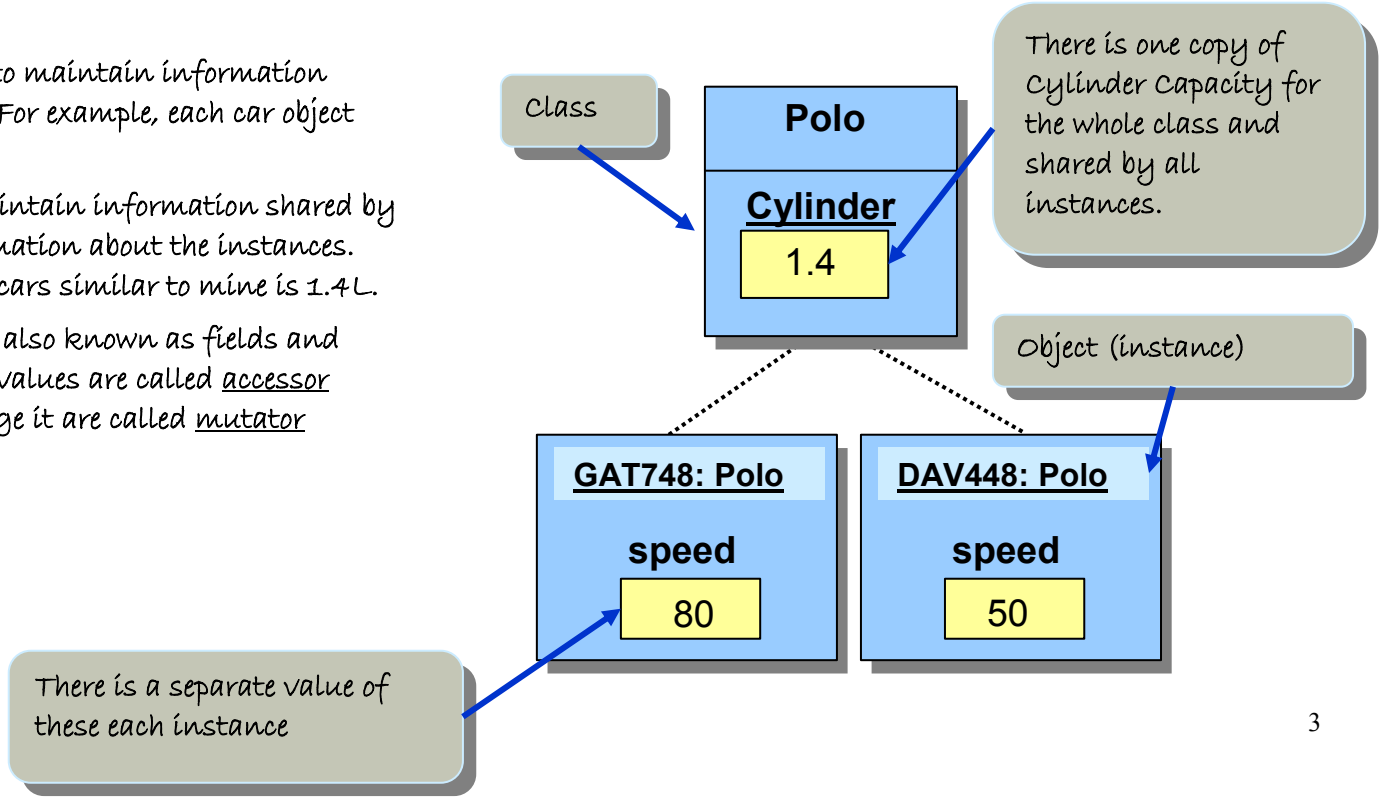
•An instance data value is used to maintain information specific to individual instances. For example, each car object maintains its speed.

•A class data value is used to maintain information shared by all instances or aggregate information about the instances. Say the cylinder capacity of all cars similar to mine is 1.4L.

Methods act on these data values also known as fields and methods which access these data values are called accessor methods whilst those which change it are called mutator methods.

Methods to use the Object

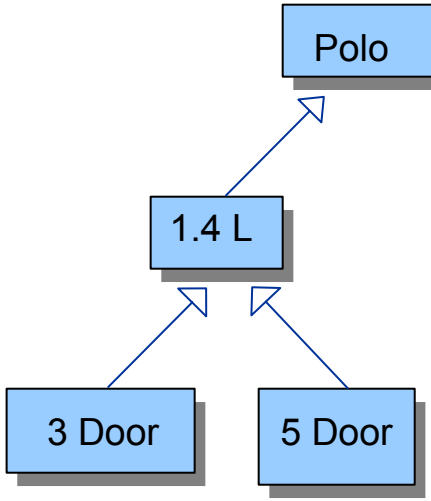
Class & Instance Data Values



Inheritance

Although there are many cars similar to my car, some cars although nearly the same as mine, differ slightly in some specific aspects. Using our analogy there may be another version of the VW polo where the cylinder capacity is 2L and has5 doors instead of three. To cater for this situation we define a super class Polo which contains all common features shared amongst all Polos and then we extend these features in a subclass with features which belong to only one class of that superclass. The subclass would then contain all features of the superclass and its own features.

We also call the superclass an ancestor and the subclass a descendant.

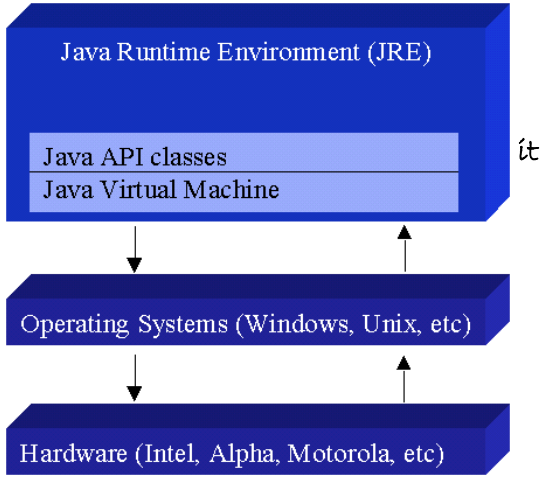


Java's Past and Present

The Java language was originally created to solve the problems surrounding personal digital assistants (PDAs) and consumer electronic products, such as microwaves and toaster ovens. The language had to be robust, small, and portable. None of the programming languages available at the time (circa 1990) would fill the bill. Some were too complex to write really robust code, and none of them was portable. Code had to be recompiled for each target chip, and consumer electronics chips were proprietary and frequently updated.

The language that met all those requirements didn't exist, so the folks at Sun decided to create a new language that would. The consumer electronics market never really made use of the new language.

However, the Java team discovered that Java's design made it ideal for another purpose: programming on the Internet. In 1993, the World Wide Web was just becoming popular, with its myriad platforms and operating systems, and desperately needed a platform-independent language—so Java found a new home.



it

Java Is Platform-Independent

The Java language was designed specifically to be platform-independent. This means that programs written in Java can be compiled once and run on any machine that supports Java. So, what exactly does platform-independence mean, and how does it differ from what was available before Java?

Platform-independence refers to a program's capability to run on various computer systems without the necessity of being recompiled. A platform means a particular processor, like the Intel 80X86, the Intel Pentium, the Motorola 68XXX, or the PowerPC.

Traditional compiled programming languages are compiled to machine-level binary code that is, of course, specific to the machine or platform on which it is compiled. The advantage is that the compiled binary runs quickly because it is running in the machine's native language. The disadvantage is that a program written in a traditional language has to be recompiled to binary code for each different hardware platform before it can be run on that machine. On the other hand, whereas traditional interpreted programming languages are machine-neutral and can be run without recompilation on various platforms, they generally run much slower than compiled applications.

Java has incorporated the best of both worlds. The Java team created a platform-specific layer, called the Java Virtual Machine (Java VM), which interfaces between the hardware and the Java program. When you install a Java-capable browser on your computer, for example, a copy of the Java interpreter is also installed, which is what enables your browser to execute Java applets and your system to run standalone Java programs.

Java interpreter and Java runtime are alternative terms for the Java Virtual Machine (VM).

The Java VM presents the same interface to any applets that attempt to run on the system, so to applets, all machines look the same. The applet itself is compiled into a form of interpretable code called Java bytecodes. This special form of code can be run on any platform that has the Java VM installed.

Bytecodes are a set of instructions that are similar to machine code but are not processor-specific. They are interpreted by the Java VM. Sun's trademark phrase "write once, run anywhere" is the promise that is fulfilled by Java. This, above all other features of the language, is what makes it so essential for interactive Web programming. It makes code much safer as it interacts only with the JVM.

Java Is Object-Oriented

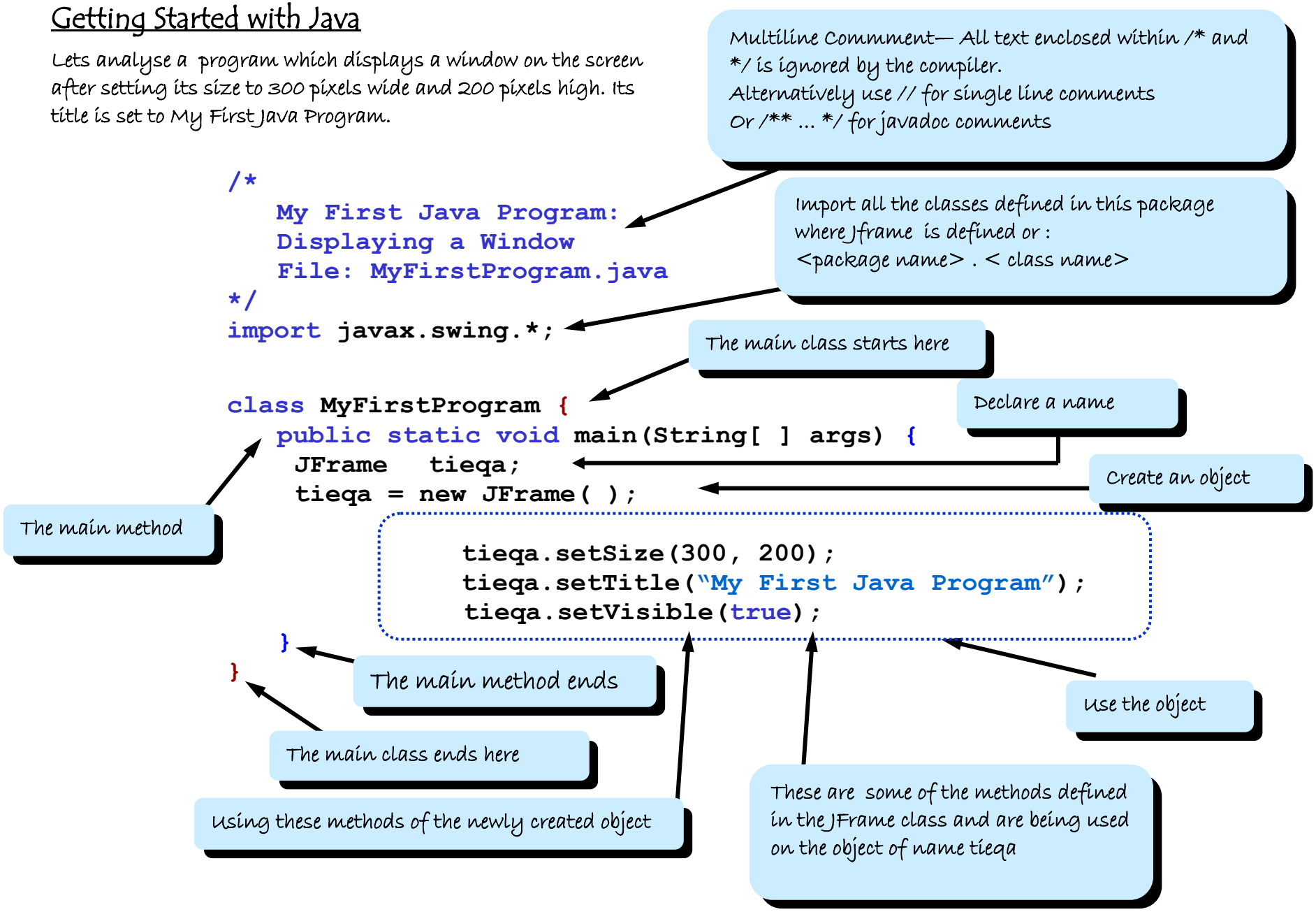
Java is a real object-oriented language, which enables you to create flexible, modular programs. It includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. It also provides networking, common Internet protocol, image handling, and user-interface toolkit functions. Java's object classes provide a rich set of functionality, and they can be extended and modified to create new classes specific to your programming needs.

Java Is Easy to Learn

Among the original design goals of the Java team members was to create a language that was small and robust. Because they started from scratch, they were able to avoid some of the pitfalls of other more complex languages. By eliminating things such as pointers, pointer arithmetic, and the need to manage memory explicitly, the Java development team made Java one of the easier languages to learn. However, you still can do anything in Java that you can do in traditional languages. If you have previously used a high-level object-oriented programming language, such as C++, much of Java will look familiar to you.

Getting Started with Java

Lets analyse a program which displays a window on the screen after setting its size to 300 pixels wide and 200 pixels high. Its title is set to My First Java Program.



Object Declaration

To be able to use an object, it must be given a particular identifier name. These identifiers :

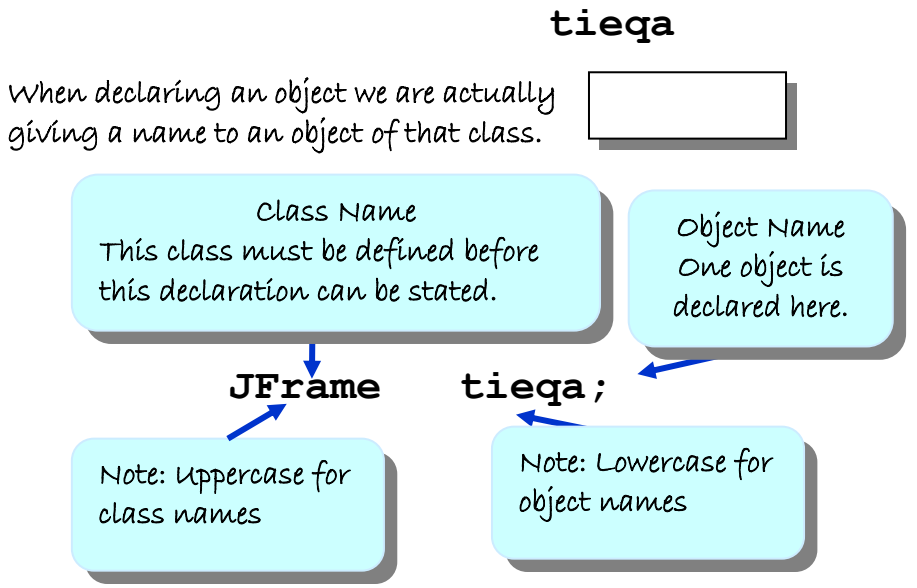
- cannot be Java keywords:

JAVA KEYWORDS.

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

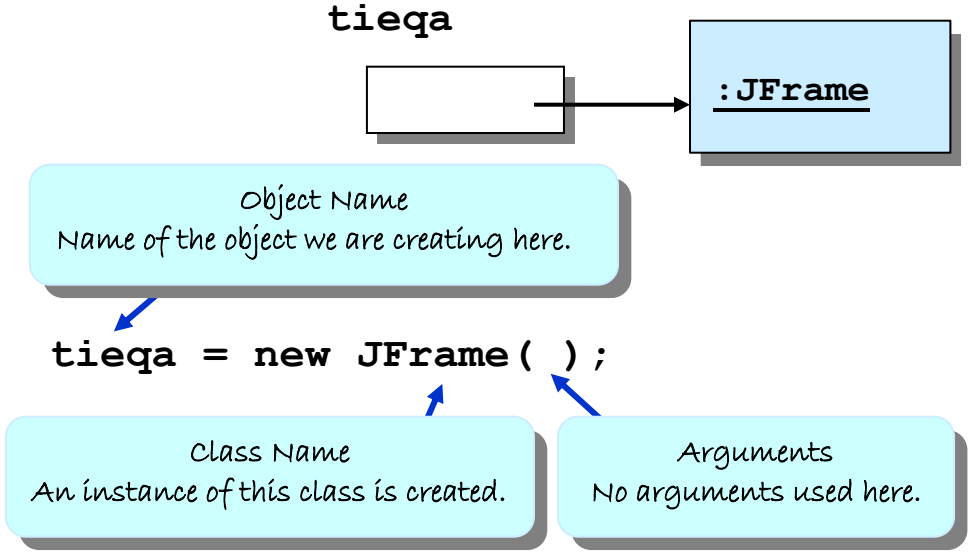
- cannot contain punctuation marks or spaces
- must not start with a numeral.

It is a common convention to name objects starting with a lower case letters and using an uppercase letter to separate words in the identifier. On the other hand the name of the classes normally start with an Uppercase letter. Java is case-sensitive, meaning that lowercase letters and upper-case letters are totally different letters.



Object Creation

When we create an object (ex tieqa = new JFramje();) we are effectively reserving memory space for an object. This is when the object really comes in existence.

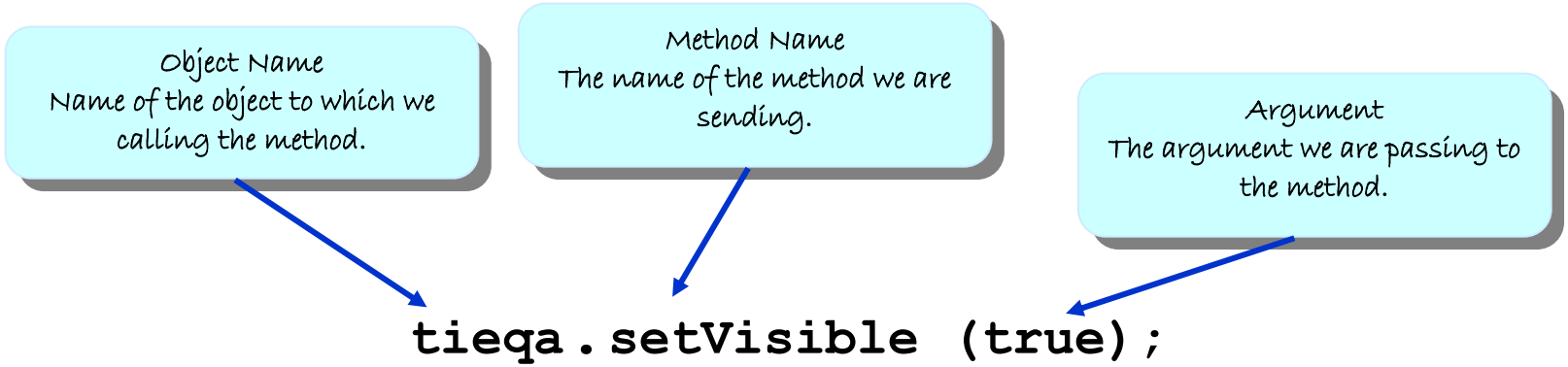


Calling Methods

In Java we apply call methods on objects. So we use the name of the object and the method we want to use of that object separated by a dot. Note since we call the method of an object we are only effecting that particular instance. (We will see later some exceptions on static classes).

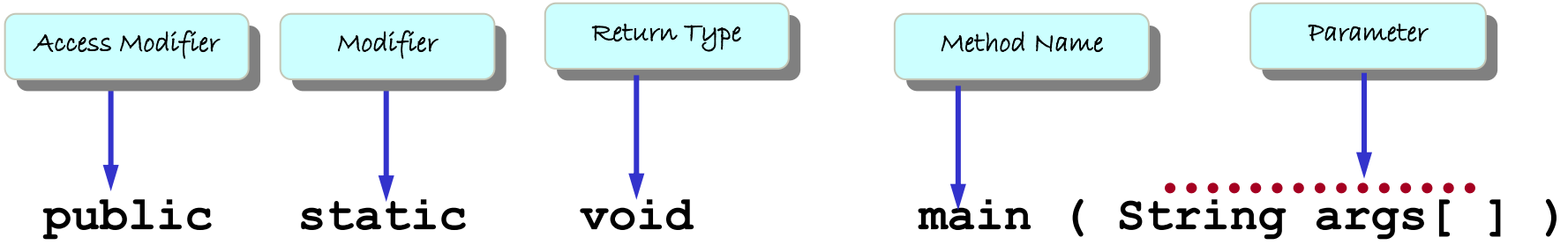
Examples:

```
account.deposit( 200.0 );
student.setName("john");
myCar.startEngine();
```



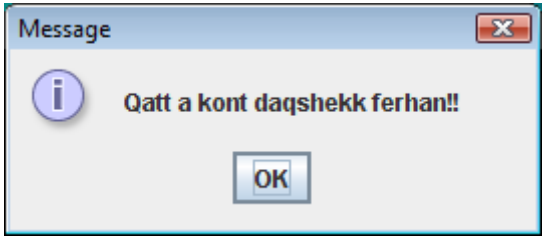
Method Declaration

When declaring a class the following statements must be included. This is the main method of which one and only one is needed for a program to execute.



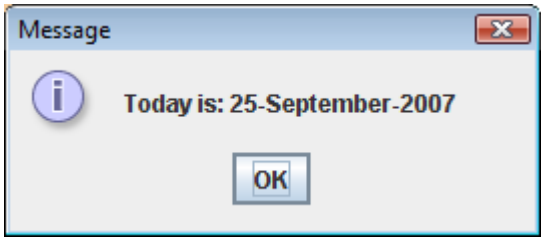
Example 1

```
/* We first import the swing package from which we are using the JOptionPane class
 * This can also be done by: import javax.swing.JoptionPane; which imports just that class instead of *
 * which means a wildcard to import all classes in the package. This has no added cost to the code.
 */
import javax.swing.*;
/**
 * Example 1:   Simple Program to demonstrate how to write the main method of a Java Program.
 *              It uses a class imported from the javax.swing package to write a message in a  window on the screen.
  * @author John Cutajar
 * @version 1.0
 */


/* First specify the name of the class in which the main method resides:
 * Remember it must have the same as the file it is stored in (Ex1.java).
 * Remember All classes start with an uppercase letter by convention
 */
public class Ex1{                          // The main class starts here
   // The heading of the main method! - Remember PSV Eindhoven
   public static void main(String args[]){     // The main method starts here
      // Use the class by calling one of its member methods (showMessageDialog)
      // methods start with a lowercase letter.
      // The method is static so there is no need to instantiate a new JOptionPane
      JOptionPane.showMessageDialog(null,"Qatt a kont daqshekk ferhan!!");
   } // main method ends here
}  // main class ends here
```



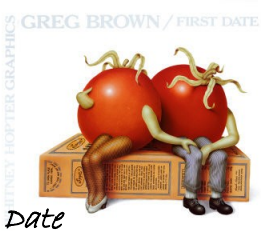
Example 2

```
import java.text.*;          //Package containing SimpleDateForamt class
import javax.swing.*;        // Package containing JOptionPane class
import java.util.*;          // Package containing Date class
/**
 * Example 2: Program to demonstrate the use of SimpleDateFormat
 * and how to obtain the current date and time in the desired format
 *
 * @author John Cutajar
 * @version 1
 */
public class Ex2{
   public static void main(String args[]){
     SimpleDateFormat sdf;
     Date  today;
     // get today's date
     today = new Date();
     // format it to the required format (long format month and year)
     sdf  = new SimpleDateFormat("dd-MMMM-yyyy");
     // Display it
     JOptionPane.showMessageDialog(null,"Today is: " + sdf.format(today));
   }
}
```



Letter	Date/Time Component	Examples
G	Era designator	AD
y	Year	1996; 96
M	Month in year	July; Jul; 07
w	Week in year	27
W	Week in month	2
D	Day in year	189
d	Day in month	10
F	Day of week in month	2
E	Day in week	Tuesday; Tue
a	Am/pm marker	PM
H	Hour in day (0-23)	0
k	Hour in day (1-24)	24
K	Hour in am/pm (0-11)	0
h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	-0800

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time



A simple Date

Variables

Variables are entities that can hold values. Before it can be used it must first be declared.

A variable has four properties:

- An Address,
- A memory location to store the value, whose storage capacity depends on the type of data to be stored,
- The type of data stored in the memory location, and
- The name used to refer to the memory location.

Numeric Data Types

There are six numerical data types: byte, short, int, long, float, and double. (Note: no inbuilt unsigned number data type)

At the time a variable is declared, it can also be initialized. For example, we may initialize the integer variables count and height to 10 and 34 as

```
Examples:

int       i, j, k;
float     numberOne, numberTwo;
long      bigInteger;
double    bigNumber;

int count = 10, height = 34;

numberOne = 3.45F;
i = 45;
```

Note: Declare the type of a variable only once, as long as it its scope hasn't ended otherwise an error will arise

Data Type	Content	Default Value†	Minimum Value	Maximum Value
byte	Integer	0	−128	127
short	Integer	0	−32768	32767
int	Integer	0	−2147483648	2147483647
long	Integer	0	−9223372036854775808	9223372036854775807
float	Real	0.0	−3.40282347E+38‡	3.40282347E+38
double	Real	0.0	−1.79769313486231570E+308	1.79769313486231570E+308

Assignment Operators

The assignment operator in Java is denoted by a single equals sign: The variable on the left of the equals sign takes the resulting value of the evaluation of the expression on the right.

The syntax is

```
<variable> = <expression> ;
```

Examples:

```
sum = firstNumber + secondNumber;
avg = (one + two + three) / 3.0;
```

In Java the overall result of the RHS expression is returned, allowing the LHS to be assigned too as follows:

This is permissible in Java

```
int x,y,z;
x = y = z = 3;
```

Other Assignment Operators

In Java there are other assignment operators :

```
+=      -=      *=      /=      %=
&=      |=      ^=
<<=     >>=
```

In each of these:

<expression1> <op>= <expression2>

Is equivalent to

<expression1> = <expression1> <op> <expression2>

```
a += 27;
```

```
a = a + 27;
```

```
i *= j + 2;
```

```
i = i * (j + 2);
```

integer division

real division

Operation	Java Operator	Example	Value (x = 10, y = 7, z = 2.5)
Addition	+	x + y	17
Subtraction	-	x - y	3
Multiplication	*	x * y	70
Division	/	x / y	1
		x / z	4.0
Modulo division (remainder)	%	x % y	3

Type Casting

What will be the data type of the following expression?

```
float x;
int y;
      x*y
```

The answer is float.

The above expression is called a mixed expression. The data types of the operands in mixed expressions are converted, based on the promotion rules.

The promotion rules ensure that the data type of the expression will be the same as the data type of an operand whose type has the highest precision.

Promotion Rules:

- if either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, both operands are converted to type int.

"My boss only notices me when I make a mistake. If I stay home, I won't make any mistakes and I'll finally get a promotion!"

Explicit Type Casting

Instead of relying on the promotion rules, we can make an explicit type cast by prefixing the operand with the data type using the following syntax:

(<data type>) <expression>

```
Example
      (float) x / 3
      (int) (x / y * 3.0)
```

Type cast the result of the expression x / y * 3.0 to int.

Type case x to float and then divide it by 3.

Implicit Type Casting

Consider the following expression:

```
double x = 3 + 5;
```

The result of 3 + 5 is of type int. However, since the variable x is double, the value 8 (type int) is promoted to 8.0 (type double) before being assigned to x.

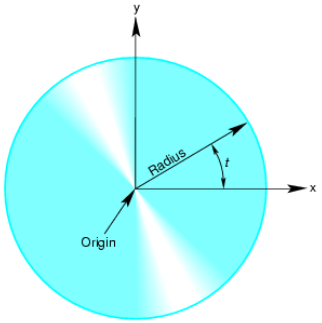
•Notice that it is a promotion. Demotion is not allowed.

```
int x = 3.5;
```

A higher precision value cannot be assigned to a lower precision variable.

Example 3

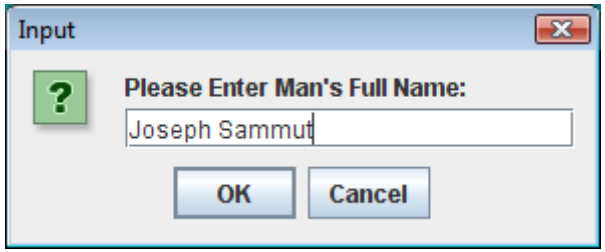
```
/**
 * Example 3: To create a new class Circle
 * with the radius as data member, A Constructor and
 * four member methods to set the Radius
 * and get radius, area and circumference
 *
 * @author John Cutajar
 * @version 1
 */
// Note this is not the main class
// Store this file in Circle.java
public class Circle
{
   // data member accessible only from within the class (private)
   private double radius;

   // default parameter-less constructor which set the value of
   // the radius initially to zero
  // We will discuss constructors later on
   public Circle(){
      radius = 0;
     }
```

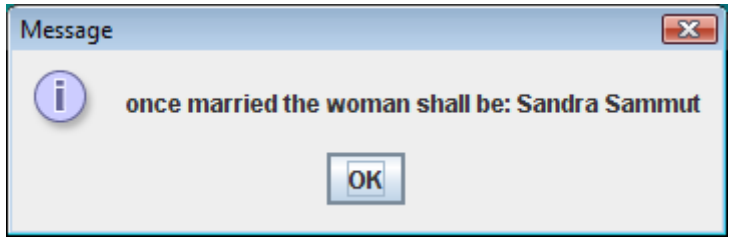



```
   // method to set the radius of the object
   public void setRadius(double r){
      radius = r;
     }

   // method to get the instance value of the radius
   public double getRadius(){
      return radius;
     }

   // method to return the value of the area of the instance
   // Note that Math.PI like most Math functions return double
   // So we have to either return double or cast it to float
   // otherwise a "Possible loss of precision" error occurs
   public double getArea(){
      return(radius*radius*Math.PI);
     }

   //method to return the circumference of the instance
   public double getCircumference(){
      return(2*radius*Math.PI);
     }
}
```

Example 4

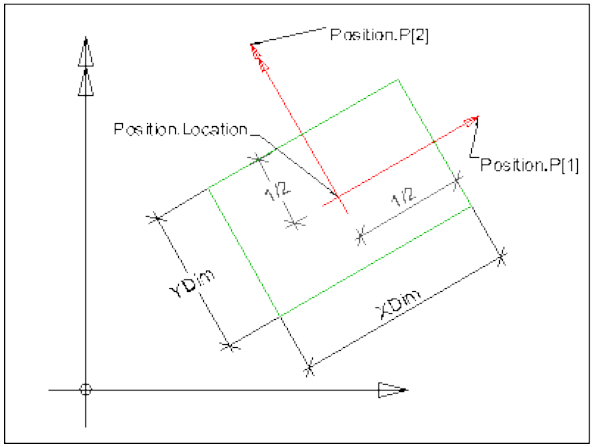
```
Textimport javax.swing.*;
/**
 * Example 4: Use of String functions to derive the woman's name
 * after marriage
 * @author John Cutajar
 * @version 1
 */
public class Ex4
{
   public static void main (String args[]){

      // variables to hold man's details
      String maleFullName, maleSurname;
      // variables to hold woman's details
      String femaleFullName, femaleName;

      // what seperates two words from each other
      String space = " ";

      maleFullName =JOptionPane.showInputDialog
                        (null,"Please Enter Man's Full Name:");
      femaleFullName =JOptionPane.showInputDialog
                        (null,"Please Enter Woman's Full Name:");

            //Extract man's surname
      maleSurname = maleFullName.substring
(maleFullName.indexOf(space)+1, maleFullName.length());

      //Extract woman's name
      femaleName = femaleFullName.substring
(0,emaleFullName.indexOf(space));

      //display married's full name
      JOptionPane.showMessageDialog(null, "once married the
woman shall be: "+femaleName+" "+maleSurname);
   }
}
```

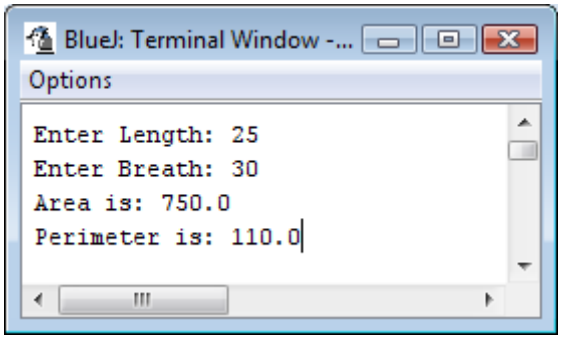




Example 5

```
import java.util.*;
/**
 * Example 5: To calculate and display the
 * area and perimeter of a rectangle
 *
 * @author John Cutajar
 */

public class Ex5{
   public static void main (String args[]){

      //declare variables
      float length, breath, area, perimeter;

      //declare a new scanner for keyboard entry
      Scanner read;

      // read length of rectangle
      System.out.print("Enter Length: ");
      read = new Scanner(System.in);
      length = read.nextFloat();

      //read breath of rectangle
      System.out.print("Enter Breath: ");
      read = new Scanner(System.in);
      breath = read.nextFloat();

      //Calculate area and perimeter
      area = length * breath;
      perimeter = 2*(length + breath);

      //Display results
      System.out.print
            ("Area is: " + area + "\n" + "Perimeter is: " + perimeter);
   }
}
```





Boolean & Char

The other two primitive data types in Java are:

boolean: The boolean data type has only two possible values: true or false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

char: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Enumerated Data Types

pAn enum type is a kind of class definition. The possible enum values are listed in the curly braces, separated by commas. By convention the value names are in upper case.

```
public enum Shape { RECTANGLE, CIRCLE, LINE }
```

Constants

We can change the value of a variable. If we want the value to remain the same, we use a constant. To specify the exact type of literal constants, one can use the L,F and D or l,f and d to define the precision.

```
Examples:
final double PI = 3.14159D;
final int    MONTH_IN_YEAR    = 12;
final short  FARADAY_CONSTANT = 23060;
```

The reserved word final is used to declare constants.

These are constants, also called *named constants*

These are called *literal constants*

Strings

A string constant consists of a sequence of characters separated by double quotes.

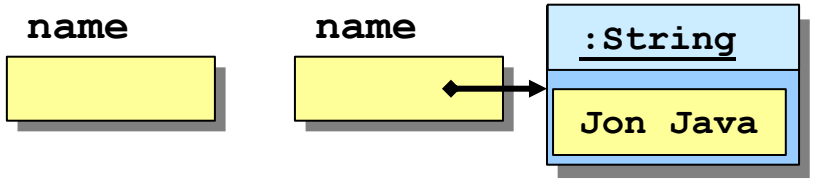
There are close to 50 methods defined in the String class.

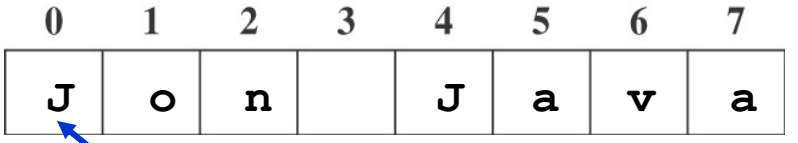
```
String   name;
name  =  new String("Jon Java");
```

Note that a String is not a primitive data type but an class, and likewise behaves like one. We say that Strings are immutable (cannot change value).

The identifier name is declared and space is allocated in memory.

A String object is created and the identifier name is set to refer to it.

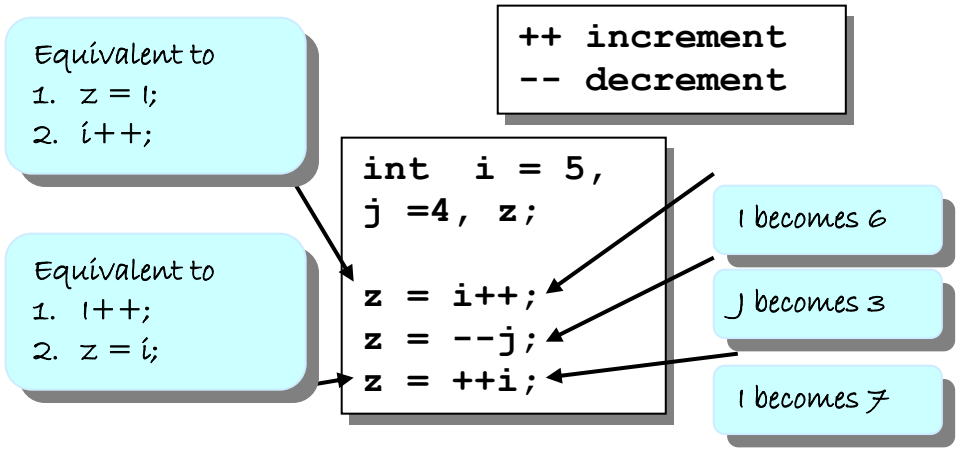




The position, or index, of the first character is 0.

Increment and Decrement

Java has two special operators for adding and subtracting one from a variable. These may either be prefix (before the variable) or postfix (after the variable).



Comparison Operators

All these operators return a boolean value of true or false, which can even be assigned to a boolean variable., or tested.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	is equal to
!=	is not equal to

Logical Operators

These operators too return a boolean value of true or false. Evaluation is from left to right and short circuit, which means if the operation is an "and" and the first clause evaluates to false, the rest of the clauses are not evaluated. If the operation is an "or" and the first clause is true the rest are omitted.

&&	and
||	or
!	not

```
Example:

int i = 10, j = 28;
boolean valid;

valid = ((i < 12) && (j <15));
```

Bitwise Logical Operators

Java provides the following bitwise operators which can be applied to any integer type:

&	bitwise and
|	bitwise inclusive or
^	bitwise exclusive or
~	one's compliment
>>	right shift
<<	left shift

Conditional Expression Operator

The conditional expression operator provides an in-line if/then/else.

- If the condition is true, the expression after the "?" is evaluated and returned as a result of the statement.
- If the condition is false, the expression after the ":" is evaluated and returned as a result of the statement.

```
int  i, j = 100, k = -1;

i = (j > k) ? j : k;
```

```
if(j > k)
     i = j;
else
     i = k;
```

```
int  i, j = 100, k = -1;

i = (j < k) ? j : k;
```

```
if(j < k)
     i = j;
else
     i = k;
```

Precedence of Operators

Java treats operators with different importance, known as precedence. There are in all 15 levels of precendence. In general, the unary operators have a higher precence over the binary operators and Parentheses can always be used to improve clarity in the expression.

```
int  j = 3 * 4 + 48 / 7;
```

Associativity of Operators

For two operators of equal precedence a second rule, "associativity" applies. Associativity is either "left to right" (left operator first) or "right to left" (righto operator first):

```
int  i = 6 * 4 / 7;
```

Operator	Associativity
() [] -> .	left to right
! ~ ++ -- - + (cast)	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= >= >	left to right
== !=	left to right
&	left to right
|	left to right
^	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= *etc*	right to left
,	left to right

Precedence & Associativity Table

Primitive and Referenced Data Types

Numerical data are called primitive data types.

Objects (including Strings) are called reference data types, because their contents is an addresses that refers to amemory locations where the objects are actually stored.

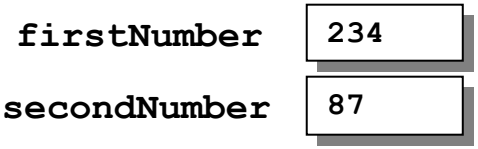
For Primitives:

```
int firstNumber;
int secondNumber;

firstNumber  = 234;
secondNumber = 87;
```

A

B

A. Variables are allocated in memory.



B. Values are assigned to variables.

If one of them say firstNumber gets reassigned the new value will overwrite the old one.

```
firstNumber = 187;
```

C

C. The value of 234 is overwritten by 187

For Objects:

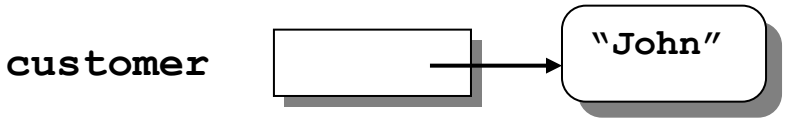
```
Customer customer, client;
customer = new Customer("John");
customer = new Customer("Mary");
```

A

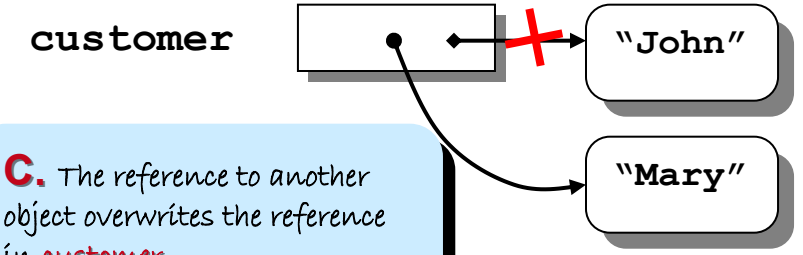
B

C

A. The variable is allocated in memory.



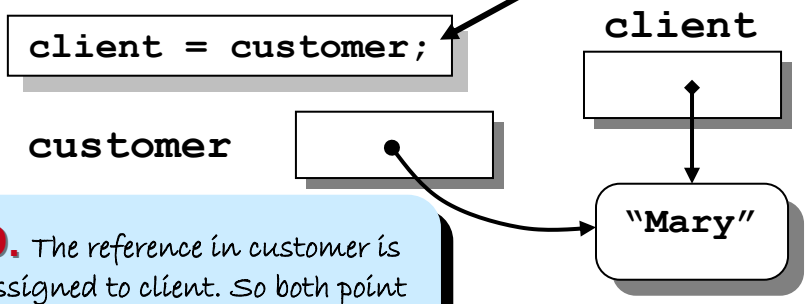
B. The reference to the new object is assigned to customer.



C. The reference to another object overwrites the reference in customer.

D

```
client = customer;
```



D. The reference in customer is assigned to client. So both point at the same object

Wrapper Classes

A wrapper is a class that contains data or an object of a specific type and has the ability of performing special operations on the data or object. Most common wrapper classes are for the primitives.

Reasons for wrapper around the primitives:

- Converting from character strings to numbers and then to other primitive data types.
- A way to store primitives in an object.

Wrapper Classes for primitive data types →

Primitive	Wrapper
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short
void	Void

```
Example:
Double myDouble = new Double("10.5");
```

Class	Method	Example
Integer	parseInt	Integer.parseInt('25') → 25 Integer.parseInt('25.3') → error
Long	parseLong	Long.parseLong('25') → 25L Long.parseLong('25.3') → error
Float	parseFloat	Float.parseFloat('25.3') → 25.3F Float.parseFloat('ab3') → error
Double	parseDouble	Double.parseDouble('25') → 25.0 Double.parseDouble('ab3') → error

Autoboxing

Autoboxing, introduced in Java 5, is the automatic conversion the Java compiler makes between the primitive (basic) types and their corresponding object wrapper classes (eg, int and Integer, double and Double, etc). The underlying code that is generated is the same, but autoboxing provides a sugar coating that avoids the tedious and hard-to-read casting typically required by Java Collections, which can not be used with primitive types.

```
Example with autoboxing:
Double myDouble = 10.5;
```

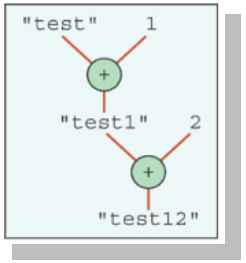
Overloading

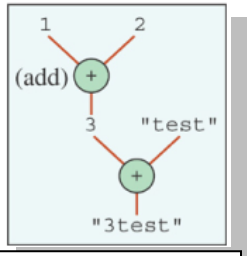
When an operation or function does different things, depending on the context it is applied to, the operator or function are said to be overloaded.

From the basic operations two operators are said to be overloaded:

1. The division is overloaded, in a sense that if you perform a division between two integers an integer division occurs, truncating all the digits after the decimal point. If one of the operands is a float or double, floating point division occurs
2. The addition operator performs the usual sum if operating between two numerals whiles concatenates the two parts if one of the operands is a String. Attention must be paid to associativity since, where in normal numeric addition the operation is commutative, in strings it's not always so.

```
output="test"+1+2;
```






```
output=1+2+"test";
```

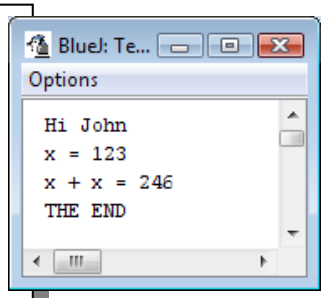
The Standard Output

Using System.out, we can output multiple lines of text to the standard output window.

We use the print method to output a value to the standard output window. The print method will continue printing from the end of the currently displayed output. We use println instead of print to skip a line.

Example:

```
int x = 123, y = x + x;
System.out.println(" Hi John");
System.out.print( " x = " );
System.out.println( x );
System.out.print( " x + x = " );
System.out.println( y );
System.out.println( " THE END" );
```



Special characters used in print which can be combined with strings.

Character	Written
backslash	\\
backspace	\b
carriage return	\r
double quote	\"
formfeed	\f
horizontal tab	\t
newline	\n
single quote	\'

The Standard Input

The technique of using System.in to input data is called standard input. We can only input a single byte using System.in directly, so to input primitive data values, we use the Scanner class (from Java 5.0).

Example:

```
import java.util.*;
...
Scanner kb;
kb = new Scanner(System.in);
int num = kb.nextInt();
```

Scanner class defined here

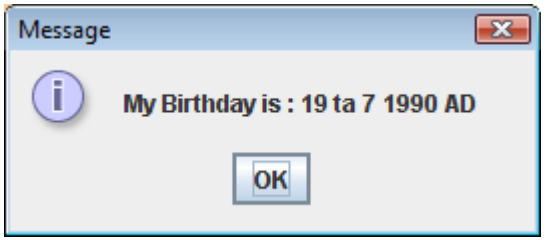
Common Scanner Methods

Method	Example
nextByte( )	byte b=kb.nextByte( );
nextDouble( )	double d=kb.nextDouble( );
nextFloat( )	float f=kb.nextFloat( );
nextInt( )	int i=kb.nextInt( );
nextLong( )	long l=kb.nextLong( );
nextShort( )	short s=kb.nextShort( );
next()	String str=kb.next();
useDelimiter(String)	kb.useDelimiter("\n");

Use this method to change the separator between inputs from a single space

Example 6

```
import javax.swing.*;
import java.util.*;


/**
 * Example on the use of the Gregorian Calendar class
 */
public class Ex9 {
   public static void main(String args[]){


      GregorianCalendar kalendarju;


      kalendarju = new GregorianCalendar(1990,7,19);


     JOptionPane.showMessageDialog(null,"My Birthday is : "
                + kalendarju.get(Calendar.DATE) + " ta "
                + kalendarju.get(Calendar.MONTH) + " "
                + kalendarju.get(Calendar.YEAR) + " AD");
   }
}
```



Example 7

```
import java.util.*;
/** Example 3: To demonstrate the use of nested for loops
 * Displays the multiplication table for a given range
 * @author John Cutajar
 */
public class Ex3
{
        public static void main (String args[]){
           Scanner read = new Scanner(System.in);
           // prompt the user to input the range
           System.out.print("Choose a Table ");
           // read the range from the keyboard
           int max = read.nextInt();
           // for every row in the table ...
           for(int j = 1; j <= max; j++){
              // for every column in the table
              for(int i = 1; i <= max ; i++){
                 // display the values on the same line
                 System.out.print(" " + i*j + " ");
                }
                // go on a new line
                System.out.println("");
           }
     }
}
```

The Math Class

The Math class in the java.lang package contains class methods for commonly used mathematical functions. Most methods of the Math class are static and therefore there is no need to instantiate any new object of the Math class before using it.

```
Example:
double      num, x, y;
x = …;
y = …;
num = Math.sqrt(Math.max(x, y) + 12.4);
```

Some of the numerous methods of the Math Class

Method	Description
abs(a)	Absolute value of a
exp(a)	Natural number e raised to the power of a.
log(a)	Natural logarithm (base e) of a.
floor(a)	The largest whole number less than or equal to a.
max(a,b)	The larger of a and b.
min(a,b)	The smaller of a and b
pow(a,b)	The number a raised to the power of b.
sqrt(a)	The square root of a.
sin(a)	The sine of a. (Note: all trigonometric functions are computed in radians) Similarly cos and tan
random()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
round(a)	Returns the closest integer value, if a is double it returns long, if float it returns int
toRadians(a)	Converts the value of a in degrees to radians

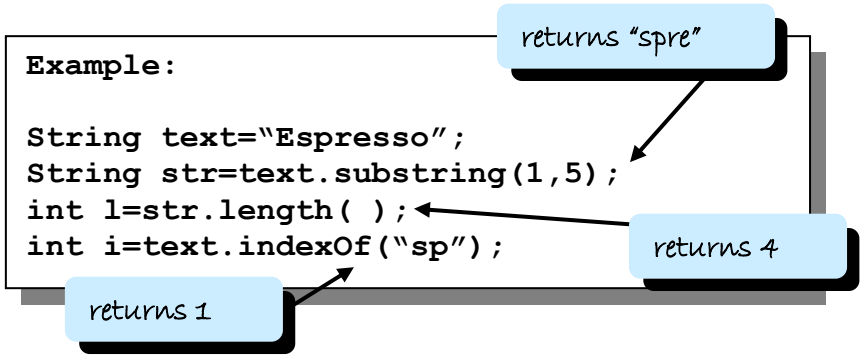
The String Class

Another very commonly used class of the java,lang package is the Sting class. Note that the java.lang package doesn't need to be imported as it is imported directly by default by java. There are various methods defined in the String class, to mention a few. For an object str of class String:

str.substring(i,j) will return a new string by extracting characters of str from position i to j-1 where 0 <= i < length of str, 0 < j <= length of str, and i <=j.

str.length() will return the number of characters in str.

str.indexOf(substr) will return the first position substr occurs in str.



Note: Strings as all objects cannot be compared using the == operator, but the equals() and compareTo() methods.

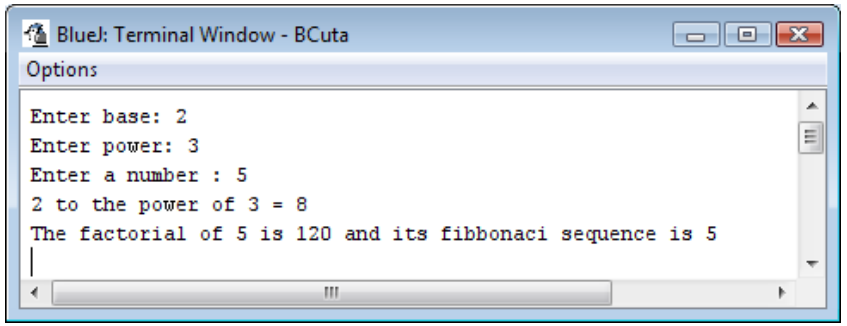
```
String s = "something", t = "maybe something else";
if (s == t)              // Legal, but usually WRONG.
if (s.equals(t))         // RIGHT
if (s > t)               // ILLEGAL
if (s.compareTo(t) > 0) // CORRECT
```

Example 8

```
/**
 * User defined class to perform some mathematical functions
 * and demonstrates also the use of recursion and static methods
 */
public class Maths{
   // calculates a to the power of b recursively
   public static long pow(int a, int b){
      if (b<=0)
         return 1;
      else
         return(a * pow(a, b-1));
      }
   //calculates n!
   public static long fac(int n){
      if (n<=1)
         return 1;
      else
         return(n * fac(n-1));
      }
   // determines the n th term of the fibonacci series
   public static long fib(int n){
      if (n<=2)
         return 1;
      else
         return(fib(n-1) + fib(n-2));
      }
   //calculates a to the power of b iteratively
   public static long power(int a, int b){
      long result = 1;
      for(int i=1; i<=b; i++)
         result *= a;
      return result;
   }
   // finds the minimum between i and j
   public static int min(int i, int j){
      return ((i < j)? i : j);
      }
   //finds the maximum between i and j
   public static int max(int i, int j){
      return ((i > j)? i: j);
   }
   // gives a random number between 0 and i
   public static int random(int i){
      return((int)((Math.random()*(i-1)+1)));
   }
   // gives a random number between i and j
   public static int random(int i, int j){
      return((int)((Math.random()*(j-i)+i)));
   }
}
```

Example 9

```
import java.util.*;
/**
 * Example Calculator: Use of the user defined Maths
 * instead of the java designed Math
 */
public class Calculator
{
        public static void main (String args[]){
           // create a new keyboard scanner
           Scanner kb = new Scanner(System.in);


           //get parameters
           System.out.print("Enter base: ");
           int base = kb.nextInt();
           System.out.print("Enter power: ");
           int power = kb.nextInt();
           System.out.print("Enter a number : ");
           int n = kb.nextInt();


        //use the user defined class (must be in the same project)
           long r = Maths.pow(base, power);
           long fac = Maths.fac (n);
           long fib = Maths.fib (n);

           //Display results
           System.out.println
                   (base + " to the power of " + power + " = " + r);
           System.out.println("The factorial of " + n + " is " +
fac + " and its fibbonaci sequence is " + fib );
           }
}
```

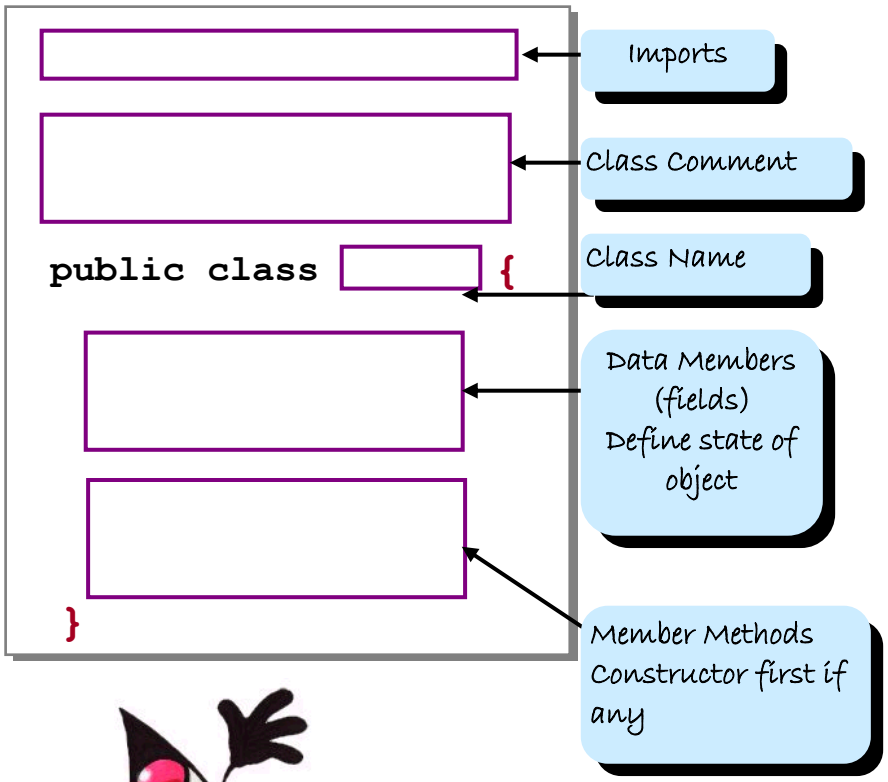


Creating a new Class

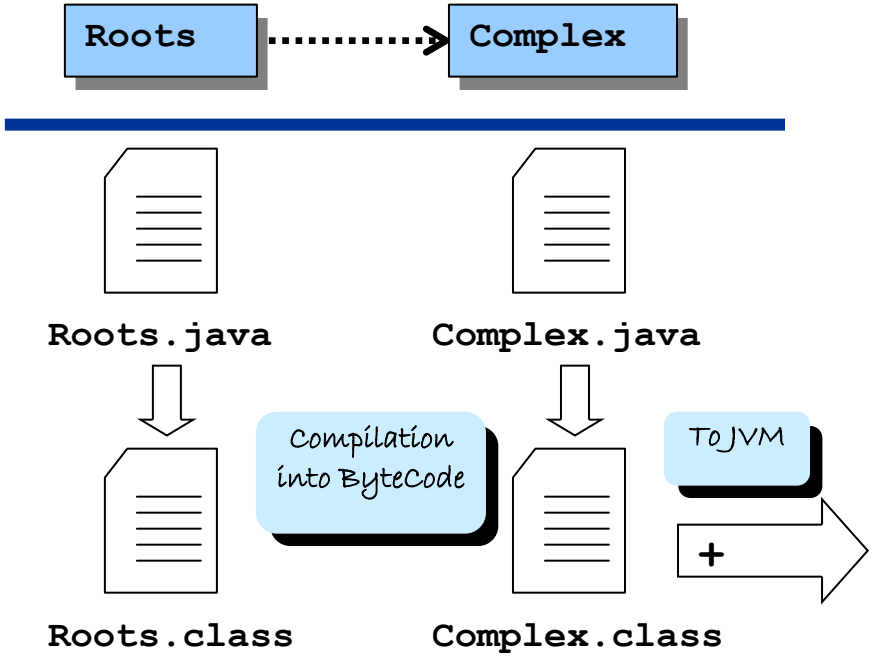
Learning how to define our own classes is the first step toward mastering the skills necessary in building large programs. Classes we define ourselves are called programmer-defined classes.

Template for non-main Classes

We have already seen how the main class looks like. Now we shall investigate how other programmer defined classes are made:



The Program Structure and Source Files



There are two source files. Each class definition is stored in a separate file.

To run the program:

1. **write Complex.java**
2. **compile Complex.java**
3. **write Roots.java**
4. **compile Roots.java**
5. **execute Roots.class**

Example 10 Bicycle Class

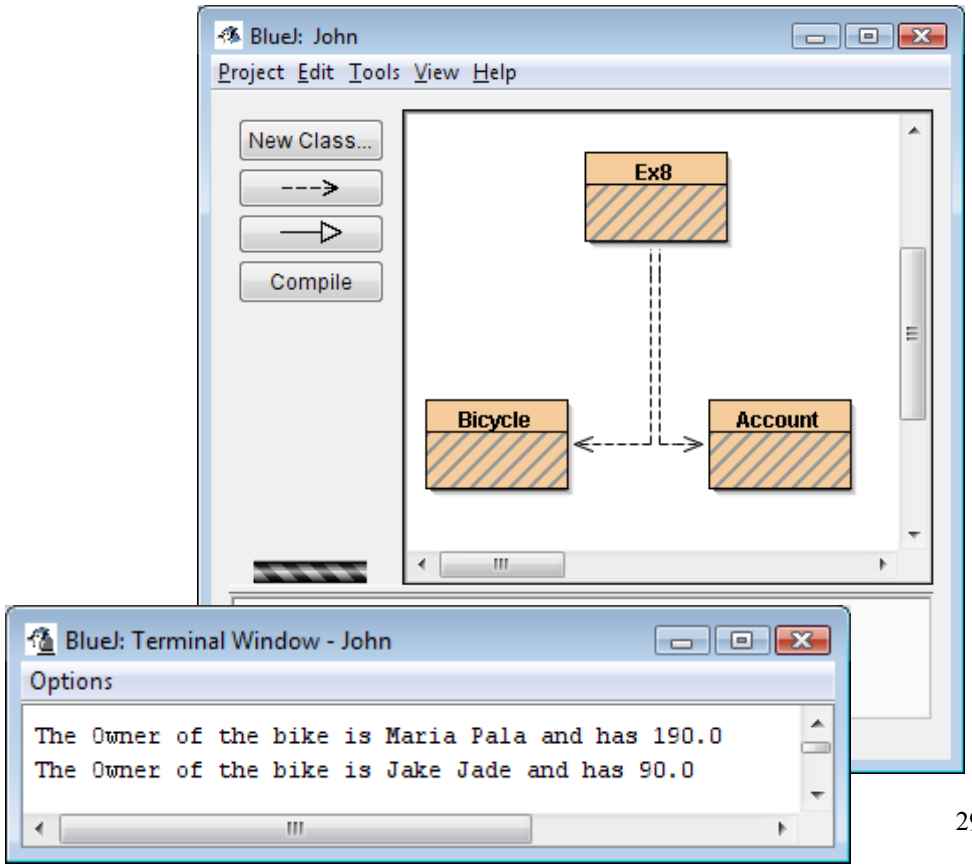
```
/**
 * Class to represent a bicycle
 */

public class Bicycle{

        //private data member
        private String owner;

        // Constructor
         public Bicycle(){
            owner = "Unknown";
           }

        //accessor method (getter)
        public String getOwner(){
            return owner;
           }

        //mutator method (setter)
        public void setOwner(String name){
            owner = name;
           }
}
```

Example 10 Account Class

```
/**
 * Class to represent an account of a person
 */
public class Account
{
        //private data member
         private double balance;

         // Constructor of the objects
         public Account(){
            balance = 100d;
           }
         //add to the balance a given amount
        public void add (double amount){
            balance += amount;
           }
         //deduct a certain amount from the balance
         public void deduct(double amount){
            balance -= amount;
           }
         // enquire the balance
         public double getBalance(){
           return balance;
           }
}
```

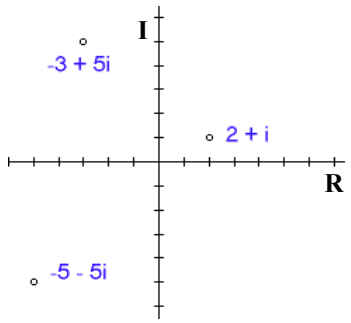
Example 11

```
/**
 * Example 11: To demonstrate the use of classes and methods
 * which use user defined classes
 */


public class Ex8
{
        public static void main (String args[]){

            // create two new bikes
            Bicycle bike1,bike2;
            bike1 = new Bicycle();
            bike2 = new Bicycle();

            //and two new accounts
            Account account1, account2;
            account1 = new Account();
            account2 = new Account();

            //set the details of bike and account 1
             bike1.setOwner("Maria Pala");
            account1.add(90.0);

            // Set details of buke and account 2
            bike2.setOwner("Jake Jade");
            account2.deduct(10.0);


            System.out.println("The Owner of the bike is " +
bike1.getOwner() + " and has " + account1.getBalance());
            System.out.println("The Owner of the bike is " +
bike2.getOwner() + " and has " + account2.getBalance());
        }
}
```



Example: Complex Numbers

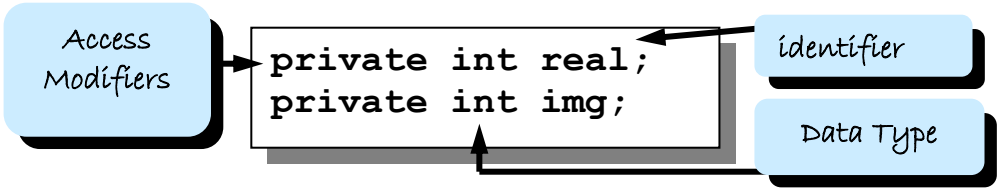
Suppose we want to create a new data type in the form of a class to represent complex numbers. A complex number consists of two parts a real part and an imaginary part. To keep things simple we shall represent these as integers.



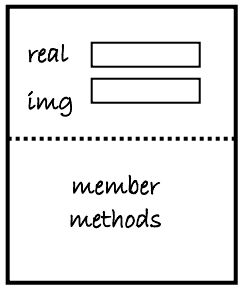
The Data Members

First of all we need to define the data required for the class, namely two integers, one for the real part and another for the imaginary part. It is good practice to declare these as private, so that they are not directly accessible from outside the object. Methods on the other hand are declared as public so that the users of this class can communicate with them.

So we begin by declaring the two required fields:

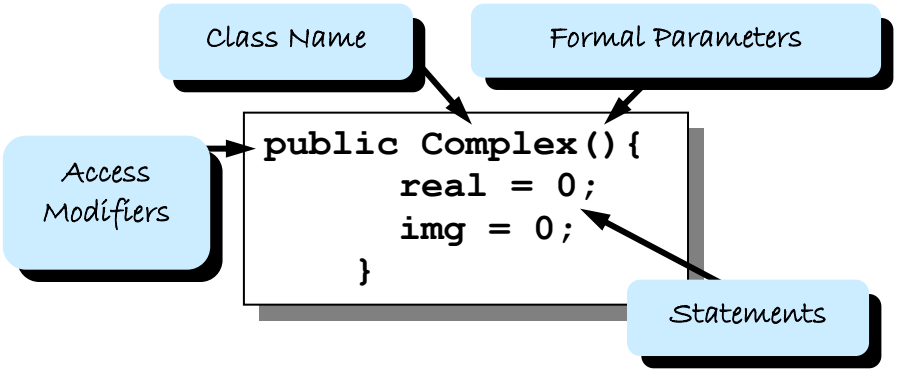


Since only methods within the object have access to these, any bug within their value can be attributed only to the member methods.



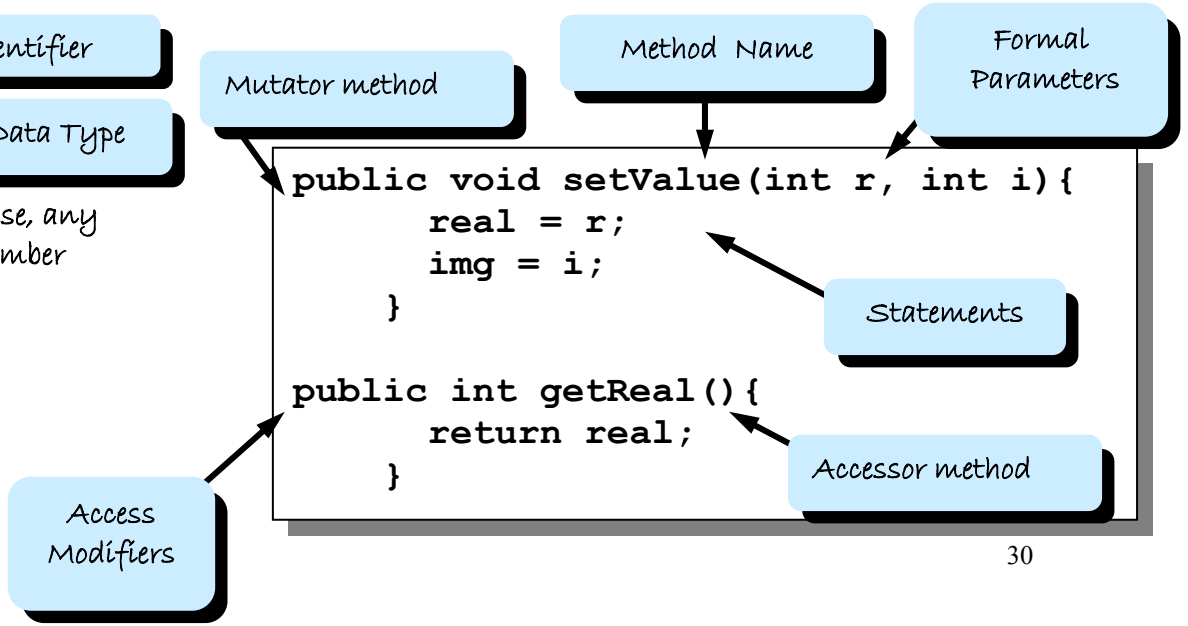
The Constructor

The constructor is a special member method which initialises an object when it is created. We limit for the time being to set all fields to zero. The constructor has always the same name as the class and has no return type:



The Member Methods

These are methods that communicate with the external environment and act upon the data members of the object.



Complex Number Implementation

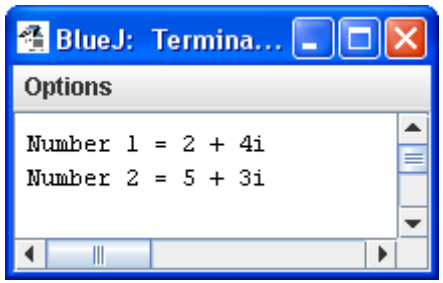
```
/**
 * Class to represent a complex number
 * @author John Cutajar
 * @version Example1
 */

public class Complex
{
  // The Real Part of the Complex Number
  private int real;

  // The Imaginary Part
  private int img;

  // The Constructor (Initialiser)
  public Complex(){
      real = 0;
      img = 0;
    }

  // Set the value of the complex number
  public void setValue(int r, int i){
      real = r;
      img = i;
    }

  // Get the real value
  public int getReal(){
      return real;
    }

  // Get the imaginary part
  public int getImaginary(){
      return img;
    }
}
```

```
/**
 * Main Class of Example 1
 * @author John Cutajar
 * @version Example1
 */
public class Roots{

   public static void main (String args[]){

      Complex number1;
      number1 = new Complex();

      Complex number2 = new Complex();

      number1.setValue(2,4);
      number2.setValue(5,3);

      System.out.println("Number 1 = " +
                         number1.getReal() + " + " +
                         number1.getImaginary()+"i");
      System.out.println("Number 2 = " +
                         number2.getReal() + " + " +
                         number2.getImaginary()+"i");
   }

}
```

Creating a new instance of Complex

Calling the setValue Method

Calling the the other Methods



Parameters

An argument or actual parameter is a value we pass to a method. A parameter or formal parameter is a placeholder in the called method to hold the value of the passed argument.

```
public class Complex{
...
// Set the value of the complex number
  public void setValue(int r, int i){
       real = r;
       img = i;
    }
...
}
```

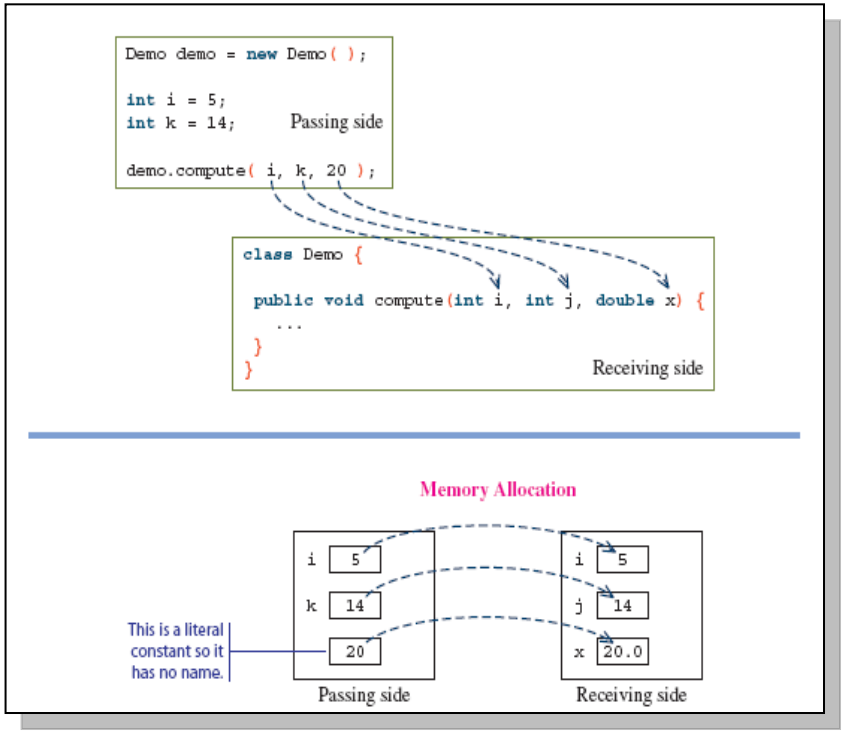
Formal Parameters

```
public class Roots{

   public static void main (String args[]){
     ...
     Complex number2 = new Complex();

     number2.setValue(3,5);
```

Actual Parameters

Matching Actual and Formal Parameters

- The number or arguments and the parameters must be the same
- Formal and actual parameters are paired left to right
- The matched pair must be assignment-compatible (e.g. you cannot pass a double argument to a int parameter)
- When we are passing primitive data values, the passing is by value, so the receiving side cannot alter the values for the passing side.



Pass-by-value

The actual parameter (or argument expression) is fully evaluated and the resulting value is copied into a location being used to hold the formal parameter's value during method/function execution. That location is typically a chunk of memory on the runtime stack for the application (which is how Java handles it), but other languages could choose parameter storage differently.

Pass-by-reference

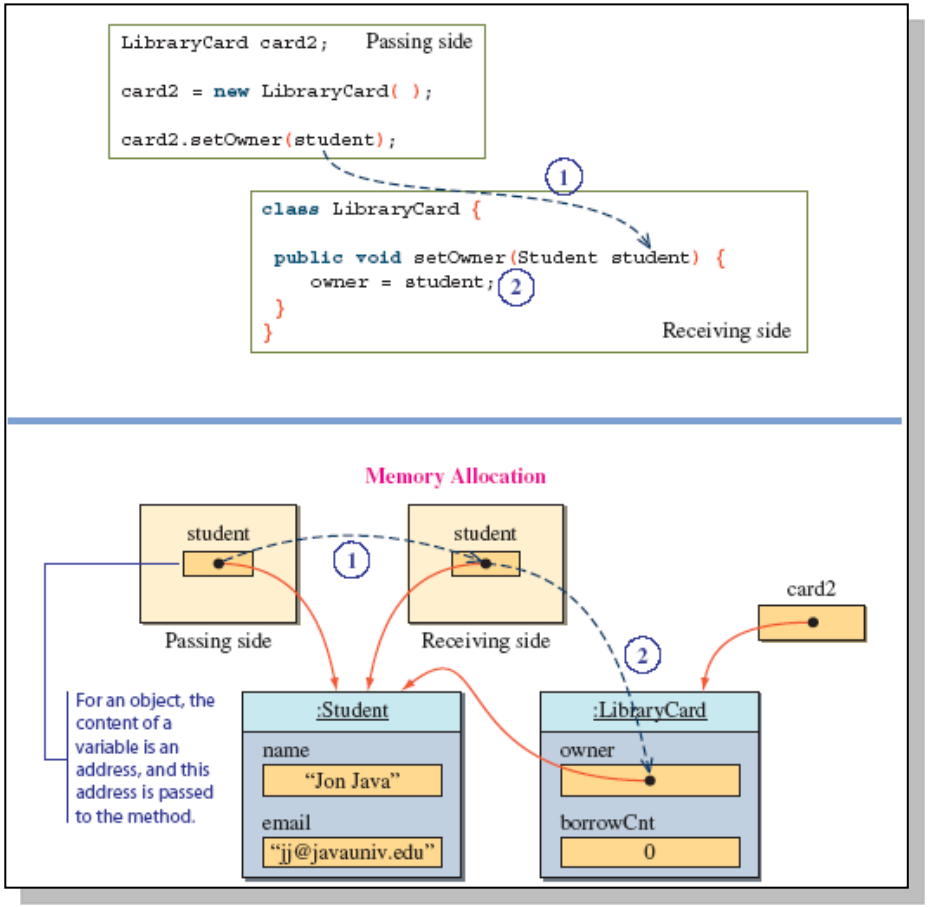
The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.

Java is strictly pass-by-value

Passing Objects as Parameters

As we can pass int and double values, we can also pass an object to a method. When we pass an object, we are actually passing the reference (address) of an object.

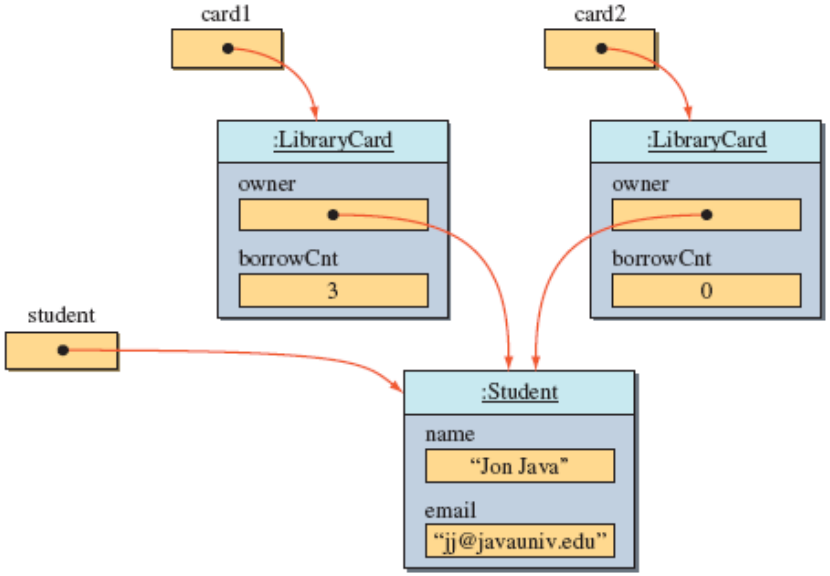
It means a duplicate of an object is NOT created in the called method



Sharing an Object

```
Student     student;
LibraryCard card1, card2;

student = new Student( );
student.setName('Jon Java');
student.setEmail('jj@javauniv.edu");

card1 = new LibraryCard( );
card1.setOwner(student);
card1.checkOut(3);

card2 = new LibraryCard( );
card2.setOwner(student); //the same student is the owner
                         //of the second card, too
```

We pass the same Student object to card1 and card2

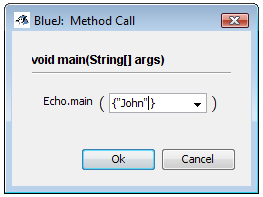


Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the

Note on Parameter Passing

1. Arguments are passed to a method by using the pass-by-value scheme.
2. Arguments are matched to the parameters from left to right. The data type of an argument must be assignment-compatible with the data type of the matching parameter.
3. The number of arguments in the method call must match the number of parameters in the method definition.
4. Parameters and arguments do not have to have the same name.
5. Local copies, which are distinct from arguments, are created even if the parameters and arguments share the same name.
6. Parameters are input to a method, and they are local to the method. Changes made to the parameters will not affect the value of corresponding arguments

Command-Line Arguments



A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.

When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings. In the following example, the command-line arguments passed to the Echo application in an array that contains a single String: "John".

```
public class Echo {
    public static void main (String[] args) {
            System.out.println(args[0]);
        }
}
```

```
java Echo John
```

Displays "John"

Organizing Classes into a Package

For a class A to use class B, their bytecode files must be located in the same directory.

> This is not practical if we want to reuse programmer-defined classes in many different programs

The correct way to reuse programmer-defined classes from many different programs is to place reusable classes in a package.

A package is a Java class library.

Creating a Package

The following steps illustrate the process of creating a package name myutil that includes the Fraction class.

1. Include the statement: **`package myutil;`**

   as the first statement of the source file for the Fraction class.
2. The class declaration must include the visibility modifier public as: **`public class Fraction { ...}`**
3. Create a folder named myutil, the same name as the package name. In Java, The package must have a one-to-one correspondence with the folder.
4. Place the modified Fraction class into the myutil folder and compile it.
5. Modify the CLASSPATH environment variable to include the folder that contains the myutil folder.

CLASSPATH

Specifying where to search for additional libraries in Windows is easily done by setting the environment variable CLASSPATH, which Java uses to see where it should find Java programs and libraries.

Information Hiding & Access Modifiers

The access modifiers public and private designate the accessibility of data members and methods. If a class component (data member or method) is declared private, client classes cannot access it. If a class component is declared public, client classes can access it. Internal details of a class are declared private and hidden from the clients. This is information hiding.

If none of this is declared, the method or data member becomes package friendly and can be accessed only from classes in the same package.

Client

```
...
Service obj = new Service();
obj.memberOne = 10;
obj.memberTwo = 20;
obj.doOne();
obj.doTwo();
...
```

```
class Service {
    public  int memberOne;
    private int memberTwo;
    public void doOne() {
    ...
    }
    private void doTwo() {
    ...
    }
}
```

Service

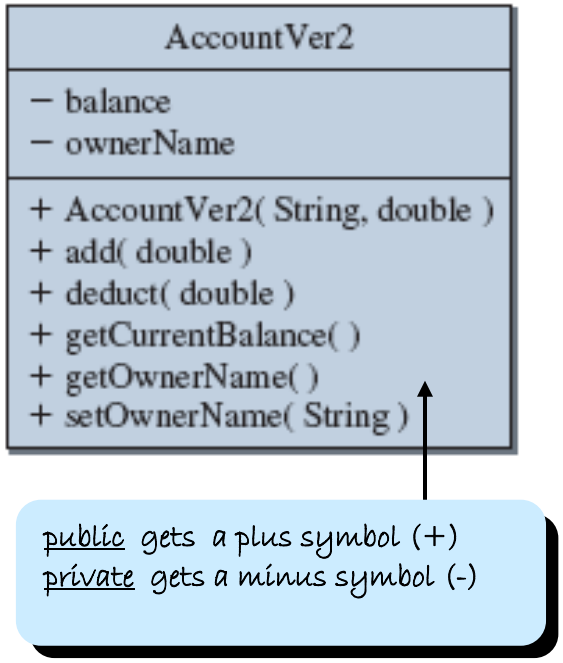
Data Members Should be Private

Data members are the implementation details of the class, so they should be invisible to the clients. Declare them private.

Exception: Constants can (should) be declared public if they are meant to be used directly by the outside methods.

Guideline for Access Modifiers

- Declare the class and instance variables private.
- Declare the class and instance methods private if they are used only by the other methods in the same class.
- Declare the class constants public if you want to make their values directly readable by the client programs. If the class constants are used for internal purposes only, then declare them private.



Static or Dynamic

There are two types of methods or variables.

Instance methods or variables are associated with an object and methods use the instance variables of that object. This is the default.

Static methods use no instance variables of any object of the class they are defined in. If you define a method to be static, you will be given a rude message by the compiler if you try to access any instance variables. You can access static variables, but except for constants, this is unusual. Static methods typically take all their data from parameters and compute something from those parameters, with no reference to variables. This is typical of methods which do some kind of generic calculation. A good example of this are the many utility methods in the predefined Math class.

Thus in any program there is only one version of a static method or variable whilst in dynamic ones there is a new version every instance created. Thus on static methods there is no need to call the new statement in the client class as in dynamic classes.

```
public static double cube(int x) {
     return (x*x*x);
}
```

```
double result = cube(3)
```

no need to create a new instance of cube

Class Constants

We illustrate the use of constants in programmer-defined service classes here.

Remember, the use of constants

- provides a meaningful description of what the values stand for. number = UNDEFINED; is more meaningful than number = -1;
- provides easier program maintenance. We only need to change the value in the constant declaration instead of locating all occurrences of the same value in the program code.

Use all UPPERS for constants

```
class Dice {
 private static final int MAX_NUMBER=6;
 private static final int MIN_NUMBER=1;
 private static final int NO_NUMBER=0;
```

final keyword declares it as constant

Class Variables

Class variables are there to keep a overall copy of a value shared among all objects of the same class, say the total number of objects created, or the sum of money gained from all objects of a vending machine.

```
Class Account{
  private static int minimumBalance;
```

Variables start with lowers, an Upper to separate words– no underscores.

Local Variables

Local variables are declared within a method declaration and used for temporary services, such as storing intermediate computation results.

```
public double convert(int num) {

    double result;

    result = Math.sqrt(num * num);

    return result;
}
```

local variable

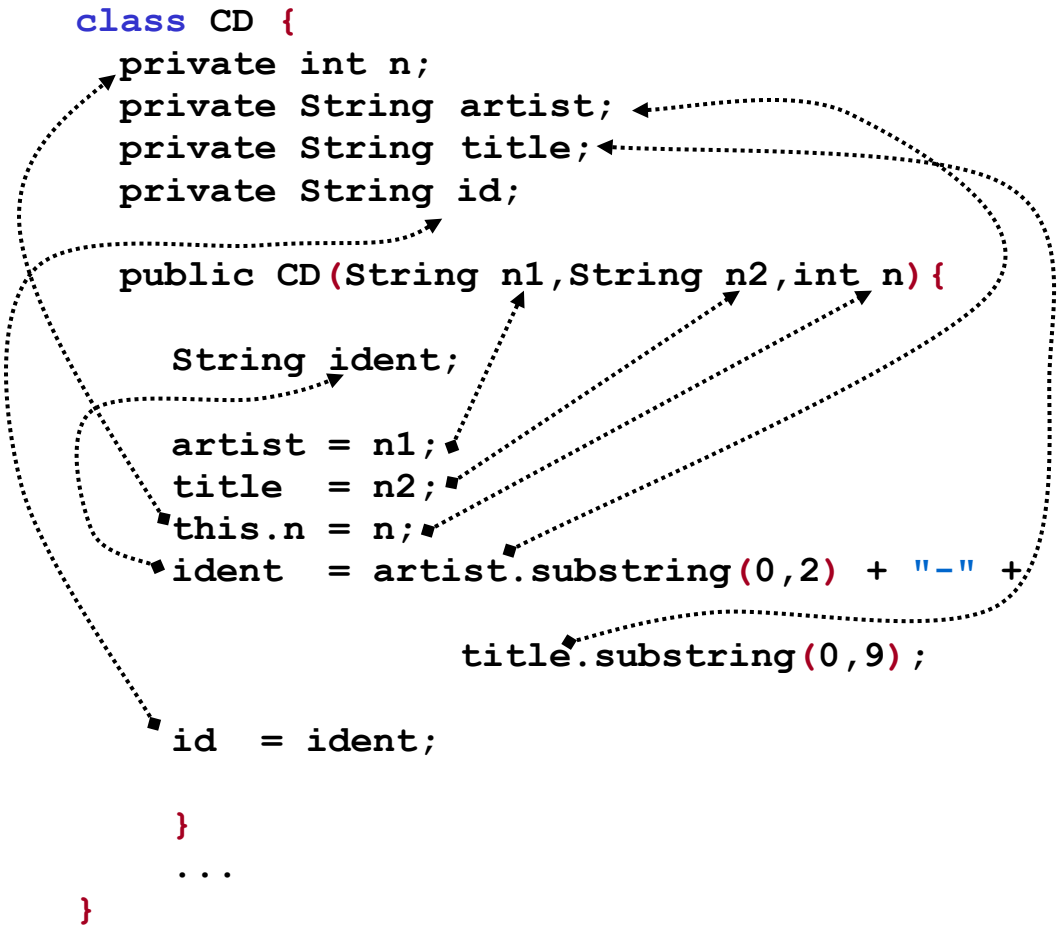
Locals, Parameters & Data Members

An identifier appearing inside a method can be a local variable, a parameter, or a data member.

The rules are:

- If there's a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
- Otherwise, if there's a matching data member declaration, then the identifier refers to the data member.
- Otherwise, it is an error because there's no matching declaration.

Variable Matching



"this" Keyword

Normally the local variable or the parameter hide the global variable. To overcome this, the keyword "this" is used to refer to the global variable.

Method Overloading

The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists.

```
public class DataArtist {
   ...
   public void draw(String s) {
            ...
   }
   public void draw(int i) {
            ...
   }
   public void draw(double f) {
            ...
   }
   public void draw(int i, double f) {
            ...
   }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

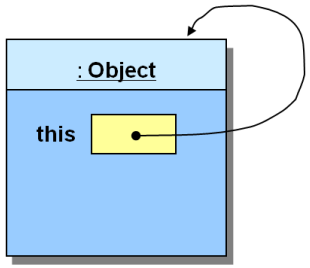
The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Constructor Overloading

The constructor of the class can be overloaded as all other methods, and it is normal practice to provide more than one constructor method.

```
Circle circle1 = new Circle(3.5d);
Circle circle2 = new Circle();

public class Circle{
     private double radius;
     ...
     public Circle(){
          radius = 0;
    }

    public Circle(double r){
          radius = r;
     }
}
```

More on the reserved word "this"



- As we have seen previously the reserved word this is called a self-referencing pointer because it refers to an object from the object's method or data item.
- This can also be used to call a different overloaded constructor of the same object..

Pay Attention: The call to this in this case must be the first statement in the constructor.

```
public Circle(){
     this(0D);
}
```

Modification to the Complex Numbers

Now that we have other tools at hand we can consider some modifications to the Complex Numbers class implementation. First we use the conditional assignment to display properly the numbers in case the imaginary part is negative and we provide an additional constructor to initialise a complex number to a given value.

```
...
Complex number1;
number1 = new Complex(3,5);
...
System.out.println("Number 1 = " +
     number1.getReal() +
     ((number1.getImaginary()>0)?"+":"") +
     number1.getImaginary()+"i");
System.out.println("Number 2 = " +
     number2.getReal() +
     ((number12.getImaginary()>0)?"+":"") +
     number2.getImaginary()+"i");
```

```
public class Complex{
    ...
    public Complex(int r, int i){
          real = r;
          img = i;
     }
     ...
}
```

The toString Method

The toString returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

So for our Complex number class the toString method would look as this:

```
public class Complex{
      ...
  // Overiding the default toString
  public String toString(){
      return(real+((img>0)?"+":"")+img+"i");
    }
    ...
}
```

This method is the method called when you try to print the object directly.

```
...
Complex number1;
number1 = new Complex(3,5);
...
System.out.println("Number 1 = "+number1);
System.out.println("Number 2 = "+number2);
```

Solution much more neat !

Example 12

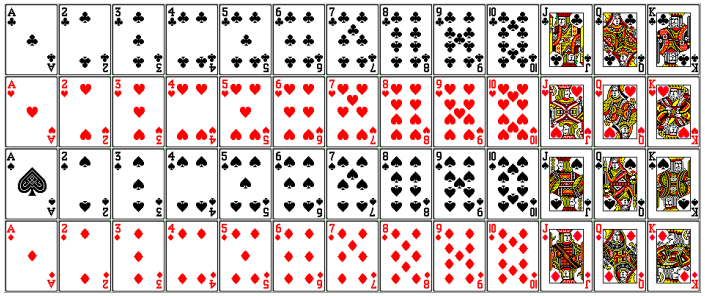
```
/**
 * Class for Cards.
 * @author John Cutajar
 */
public class Card
{
   // The suit of the card
   private String suit;
   // The number or image of the card
   private String digit;
    // Constructor which creates a new card
   public Card()
   {
   //Generate a random number from 0 to 51
   int number = (int) (Math.random()*51);
   // Use the quotient to define the siut
   int qoutient  = number/13;
   // and the remainder to define the digit
   int remainder = number%13;
```

from 0 to 12

from 13 to 25

from 26 to 38

from 39 to 51



```
   switch(qoutient)
   {
      case 0 : suit = "Clubs"; break;
      case 1 : suit = "Hearts"; break;
      case 2 : suit = "Spades; break;
      case 3 : suit = "Diamonds"; break;
   }
   switch(remainder)
   {
      case 0 : digit = "A"; break;
      case 10 : digit = "J"; break;
      case 11 : digit = "Q"; break;
      case 12 : digit = "K"; break;
      default : digit = "" + remainder;
   }
  }


   // Override the default toString method
   public String toString(){
      return digit + " of " + suit;
   }
     // Method to compare two cards: avoid the useless if
   public boolean equals(Card other){
      return((this.suit==other.suit)&&(this.digit==other.digit));
   }
}
```

Selection Statements

The if Statement

```
if ( <boolean expression> )
     <statement>
```

In the above syntax, if the Boolean expression evaluates to true, the following statement or block of statements enclosed within braces is performed, otherwise nothing is executed.

Example:

```
if ( mark < 45 )
     System.out.println("You failed");
```

The if-then-else Statement

```
if ( <boolean expression> )
     <statement A>;
else
     <statement B>;
```

If the Boolean expression evaluates to true, the statement A or block of statements enclosed within braces is performed, otherwise Statement B is performed..

Note:

- The parenthesis surround the condition. Since the "then" keyword does not exist in Java, the parenthesis separate the condition from the statement.
- If more than one statement is needed, braces are used to enclose all the statements.
- In the if-then else statement there is a semicolon even before the else only in case no braces are used.

```
if ( mark < 45 )
     System.out.println("You failed");
else {
        System.out.println("Lucky !! ");
        mark++;
      }
```

Warnings:

- Don't put a semicolon after the condition cause it means do nothing if condition is true:

```
if ( mark < 45 );
     System.out.println("You failed");
```

- Don't use the assignment (=) operator instead of the comparison operator (==)

```
if ( mark = 45 )
     System.out.println("Tkaxkart");
```

Nested ifs

The else associates with the nearest if. Thus braces must be used to impose the required association

For a condition consisting of more than one clause they must be tied by a Boolean expression:

```
int i = 100;
if (i>0)
     if (i>1000)
          System.out.println("i is big");
     else
          System.out.println("i is medium);
```

```
i is medium
```

```
int i = -20;
if (i>0)
     if (i>1000)
          System.out.println("i is big");
     else
          System.out.println("i is negative);
```

P	Q	P && Q	P || Q	!P
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Remember De'Morgan's (break the Bar change the sign)

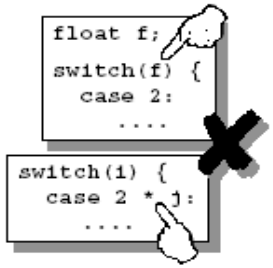
```
Rule 1: !(P && Q) ≡ !P || !Q
Rule 2: !(P || Q) ≡ !P && !Q
```

The switch Statement

The switch statement avoids a lot of nested if's

Note:

- Only integral constants may be tested
- if no condition matches, the default is executed
- if no default, nothing is done (not an error)




```
switch (c) {
     case 'a': case 'A':
          System.out.println(r*r*Math.PI);
          Break;
     case 'c': case 'C':
          System.out.println(2*r*Math.PI);
          Break;
     case 'q': case 'Q':
          System.out.println("Bye Bye");
          break;
     default:
          System.out.println("what's it?");
}
```

- the break is a must! otherwise all statements from the matched case onwards are executed

```
int i = 3;
switch (c) {
  case 3: System.out.println("i = "+3);
  case 2: System.out.println("i = "+2);
  case 1: System.out.println("i = "+1);
}
```

```
i = 3
i = 2
i = 1
```

Comparing Objects

With primitive data types, we have only one way to compare them, but with objects (reference data type), we have two ways to compare them.

1. We can test whether two variables point to the same object (use ==), or
2. We can test whether two distinct objects have the same contents.

```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1 == str2) {
     System.out.println("They are equal");
} else {
     System.out.println("They are not equal");
}
```

They are not equal

Not equal because str1 and str2 point to different String objects.

```
String str1 = new String("Java");
String str2 = str1;

if (str1 == str2) {
     System.out.println("They are equal");
} else {
     System.out.println("They are not equal");
}
```

They are equal

now they are equal

Using equals with Strings

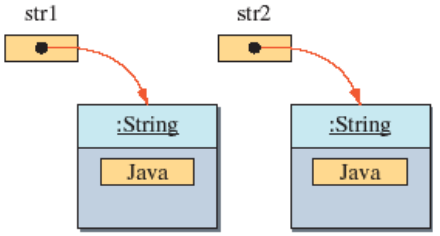
```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1.equals(str2)) {
     System.out.println("They are equal");
} else {
     System.out.println("They are not equal");
}
```

They are equal

same sequence of characters

Semantics of ==

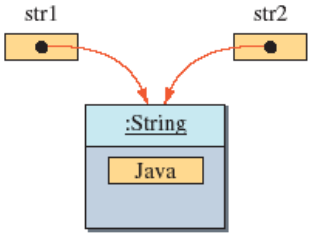
Case A: Two variables refer to two different objects.




```
String str1, str2;

str1 = new String("Java");
str2 = new String("Java");
```

str1 == str2 → false

Case B: Two variables refer to the same object.




```
String str1, str2;

str1 = new String("Java");
str2 = str1;
```

str1 == str2 → true

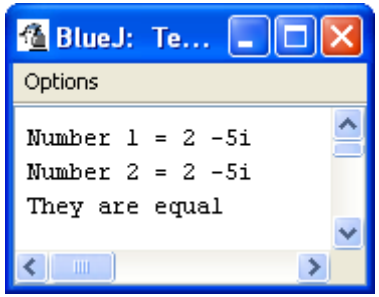
Complex Number equals

```
/**
 * Class to represent a complex number
 * @author John Cutajar
 * @version Example
 */

public class Complex
{
  // The Real Part of the Complex Number
  private int real;

  // The Imaginary Part
  private int img;

  // The Constructor (Initialiser) ...

  // Set the value of the complex number ...

  // Get the real value ...

  // Get the imaginary part ...


  // Compares two Complex numbers for equality

  public boolean equals(Complex other){
      return ((this.real==other.real)&&
                      (this.img==other.img));
  }

}
```

Useless Code:

```
if ((this.real==other.real)&&(this.img==other.img))
    return true;
    else return false;
```

```
/**
 * Main Class of Example 1
 * @author John Cutajar
 * @version Example1
 */
public class Roots{

   public static void main (String args[]){

      Complex number1;
      number1 = new Complex();

      Complex number2 = new Complex();

      number1.setValue(2,-5);
      number2.setValue(2,-5);

      System.out.println("Number 1 = "+number1);
      System.out.println("Number 2 = "+number2);
      if(number1.equals(number2))
         System.out.println("They are equal");
  }
}
```



Iterarations (Repetitive Statements)

- Repetition statements control a block of code to be executed for a fixed number of times or until a certain condition is met.
- Count-controlled repetitions terminate the execution of the block after it is executed for a fixed number of times.
- Sentinel-controlled repetitions terminate the execution of the block after one of the designated values called a sentinel is encountered.
- Repetition statements are also called loop statements.

The while statement

Syntax

```
while ( <boolean expression> )

    <statement>
```

```
int sum = 0, number = 1;

while ( number <= 100 ) {
      sum    += number;
      number++;
}
```

These statements are executed as long as number is less than or equal to 100.

Watch out!

- Watch out for the off-by-one error (OBOE).
- Make sure the loop body contains a statement that will eventually cause the loop to terminate.
- Make sure the loop repeats exactly the correct number of times.
- If you want to execute the loop body N times, then initialize the counter to 0 and use the test condition counter < N or initialize the counter to 1 and use the test condition counter <= N.
- Avoid a semicolon after the Boolean expression cause it will make it an empty loop.

```
int sum = 0, number = 1;

while ( number <= 100 ); {
      sum    += number;
      number++;
}
```

the semicolon here ends the loop immediately

```
int product =  1, number = 1,
    count   = 20, lastNumber;

lastNumber = 2 * count - 1;

while (number <= lastNumber) {
      product *= number;
      number  += 2;
}
```

The do-while Statements

Use this when you want to loop at least once.

sybtax

```
do
      <statement>
while ( <boolean expression> ) ;
```

REMEMBER!! This is not a repeat until statement. The exit condition is the same as the while

```
int sum = 0, number = 1;

do {
      sum += number;
      number++;
} while ( sum <= 1000000 );
```

These statements are executed as long as sum is less than or equal to 1,000,000.

The for Statements

syntax

```
for (<initial>;<condition>;<iteration>)
            <statement>
```

starting point | while condition | next iteration

```
for (i = 0;i < 20;i++) {
      number = scanner.nextInt();
      sum += number;
}
```

```
for (int i = 0; i < 100; i += 5)
```

i = 0, 5, 10, ... , 95

```
for (int j = 2; j < 40; j *= 2)
```

j = 2, 4, 8, 16, 32

```
for (int k = 100; k > 0; k--) )
```

k = 100, 99, 98, 97, ..., 1

The nested for

visibility only within loop

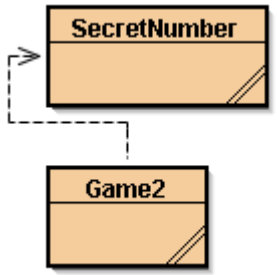
```
int price;
for (int w = 11; w <=20, w++){

      for (int l = 5, l <=25, l+=5){

       price = w * l * 19;
        System.out.print ("   " + price);
      }
    //finished one row; move on to next row
    System.out.println("");
}
```

OUTER

INNER

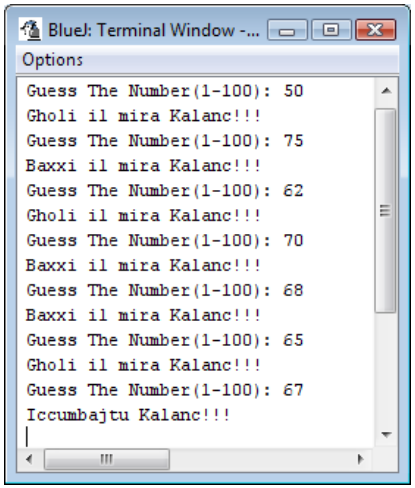
Example 13

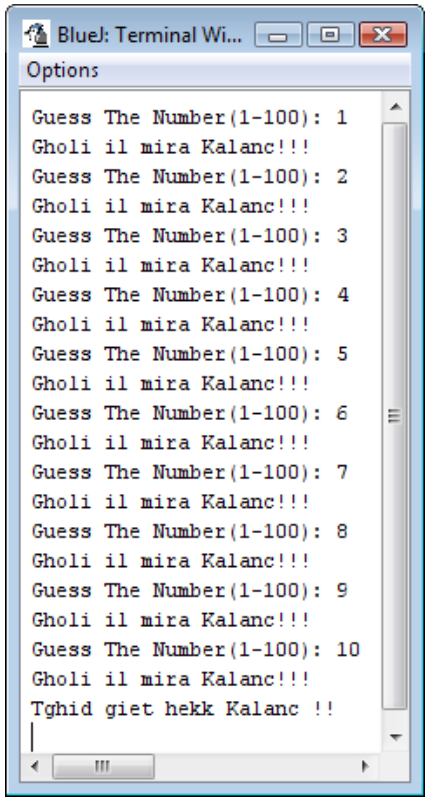
```
import java.util.*;
/**
 *Example 13 Try to guess a secret number.
 *
 * @author John Cutajar
 */
public class Game2
{
   // better define Max chances here in case I need to change it
   private static final int MAX_CHANCES = 10;

   public static void main (String args[]){

      int g;
      Scanner read = new Scanner(System.in);
      SecretNumber sigriet = new SecretNumber();
      boolean youGuessed = false;
      int chances = 1;

     do{
        // note entry on same line with print
        System.out.print("Guess The Number(1-100): ");
        g = read.nextInt();
        // Note the use of curly brackets to tie the else with the if
      if(sigriet.compare(g) ==0){
            youGuessed = true;
            System.out.println("Iccumbajtu Kalanc!!!");
         }
         else{
            chances++;
            if(sigriet.compare(g) == -1){
                System.out.println("Gholi il mira Kalanc!!!");
              }
              else{
                System.out.println("Baxxi il mira Kalanc!!!");
              }
            }
     }while ((chances<=MAX_CHANCES)&&(!youGuessed));
     if (chances > MAX_CHANCES)
         System.out.println("Tghid giet hekk Kalanc !!");
   }
}
```



Secret Number Class

```
/**
 * Class to create a secret number and reveal it to no-one.
 *
 * @author John Cutajar
 */
public class SecretNumber
{
   // this is a private Secret number
   private int secretNumber;

   // Constructor to generate a new Secret number
   public SecretNumber()
   {
      secretNumber = (int)((Math.random()*99)+1);
   }
   // Compare the given number to my own
   public int compare(int g)
   {
      if(g == secretNumber)
         return 0;
      else if(g < secretNumber)
         return -1;
      else return +1;
      }
   // check if guessed the number
   public boolean guessed(int guess)
   {
      // don't use the useless if here
      return(guess == secretNumber);
   }
}
```





Recursive Algorithms

Factorial

The factorial of N is the product of the first N positive integers:

$$N * (N-1) * (N-2) * \ldots * 2 * 1$$

The factorial of N can be defined recursively as

$$factorial(N) = \begin{cases} 1 & \text{if } N = 1 \\ N * factorial(N-1) & \text{otherwise} \end{cases}$$

A recursive method is a method that contains a statement (or statements) that makes a call to itself.

Implementing the factorial of N recursively will result in the following method.

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
public int factorial( int N ) {

    if ( N == 1 ) {
        return 1;
    }
    else {
        return N * factorial( N-1 );
    }
}
```

Generally speaking recursive algorithms turn out to be a more elegant solution when a problem can be split into a smaller problem which is convergent to some known point.

Power

Similarly x to the poer of y: $x^y = x * x^{(y-1)}$

$$power(x,y) = \begin{cases} x & \text{if } y = 1 \\ x * power(x,y-1) & \text{otherwise} \end{cases}$$

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
public int power( int x, int y ) {

    if ( y == 1 ) {
        return x;
    }
    else {
        return x * power(x,y-1);
    }
}
```

Directory Listing

List the names of all files in a given directory and its subdirectories.

Test

End case

Recursive case

```
public void directoryListing(File dir) {

  //assumption: dir represents a directory
  String[] fileList = dir.list(); //get the contents
  String dirPath = dir.getAbsolutePath();

  for (int i = 0; i < fileList.length; i++) {
     File file = new File(dirPath + "/" + fileList[i]);
     if (file.isFile()) { //it's a file
        System.out.println( file.getName() );
     } else {
        directoryListing( file ); //it's a directory
     }                            //so make a
  }                               //recursive call
}
```

When Not to Use Recursion

When recursive algorithms are designed carelessly, it can lead to very inefficient and unacceptable solutions.

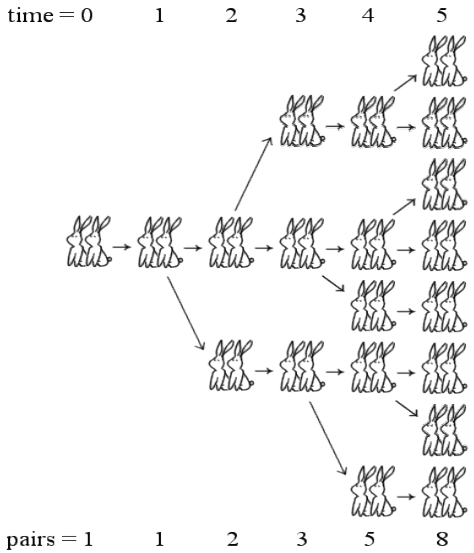
Example

For example, consider the following pattern:

Fibonacci had a pair of rabbits which took exactly one month to mature and another month to reproduce another similar pair. He wanted to know how many pairs he would have after n months. This is a classic problem that there exist no general formula to give the n th. term like other progressions.

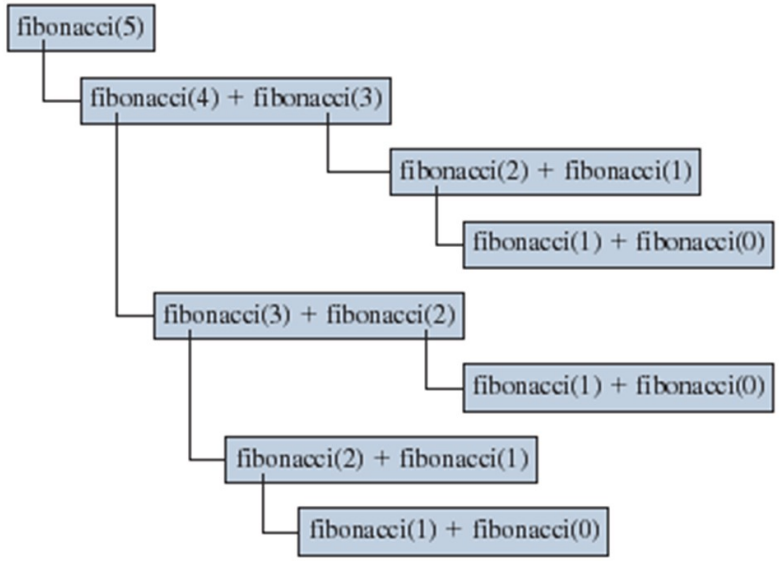
To calculate the n th. term, one can notice that all that is to be done is to add the previous two terms if n is higher than 2, otherwise it is 1.

" Que famos d'especial Miguel? - I would rather go for a marinara!!"



$$\text{fibonacci}(N) = \begin{cases} 1 & \text{if } N <= 2 \\ \text{fibonacci}(N-1) + \text{fibonacci}(N-2) & \text{otherwise} \end{cases}$$

```
public int fibonacci( int N ) {

   if (N <= 2) {
      return 1;

   } else {
      return fibonacci(N-1) + fibonacci(N-2);
   }
}
```



When to Use Recursion

In general, use recursion if

- A recursive solution is natural and easy to understand.
- A recursive solution does not result in excessive duplicate computation.
- The equivalent iterative solution is too complex.rm

Formatting

The formatter class is used to present a formatted output. Instead of using the Formatter class on its own it is much more convenient to use it a a method in the Printstream (System.out) or in the String classes.

```
System.out.printf("%6d", 498);
```

is equivalent to:

```
Formatter fmt = new Formatter(System.out);
fmt.format("%6d", 498);
```

•The general syntax is

System.out.printf(<control string>, <expr1>,<expr2>, . . .)

Some control Strings

- Integers: % <field width> d
- Real Numbers : % <field width> . <decimal places> f
- Strings: % s
- Hex: %x or %X
- Octal: %o
- Character %c
- Scientific %e or %E

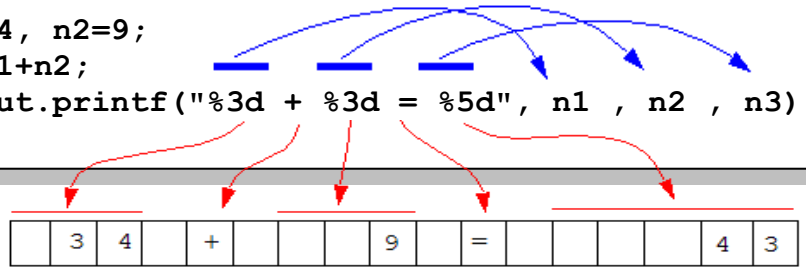
For other data types and more formatting options, please consult the Java API for the Formatter class.

When the string needs to be formatted without displaying it, the String class provides a method format which is similar to the above.

Flags: The optional flags is a set of characters that modify the output format.

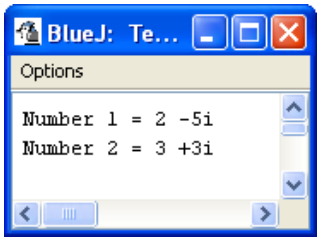
Example:

```
int n1=34, n2=9;
int n3=n1+n2;
System.out.printf("%3d + %3d = %5d", n1 , n2 , n3);
```



Some flags:

'-': left justified
'+': always include a sign
'(': negative number in brackets



Example: A better toString for Complex

```
  public String toString(){
        return(String.format("%d %+di",real,img));
}
```

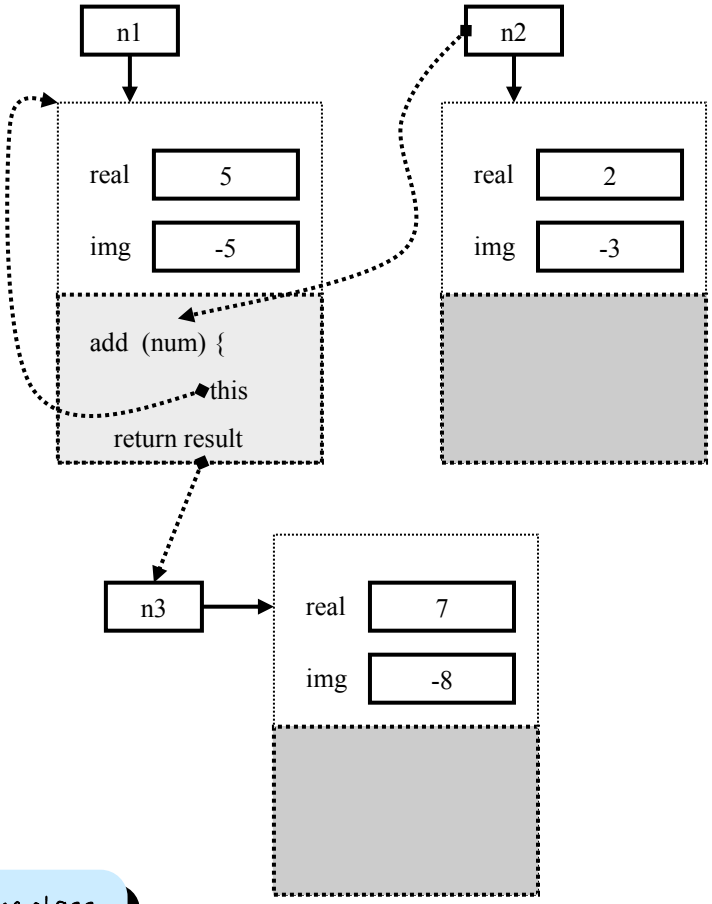
Always show sign

Returning Objects

As we can return a primitive data value from a method, we can return an object from a method also.

We return an object from a method, we are actually returning a reference (or an address) of an object.

This means we are not returning a copy of an object, but only the reference of this object



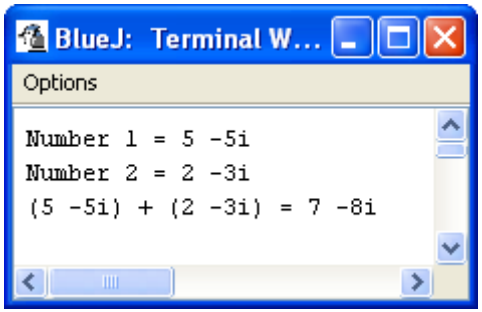

```
public class Roots
{
   public static void main (String args[]){

        Complex n1;
        n1 = new Complex(5,-5);

        Complex n2 = new Complex();
        n2.setValue(2,-3);

        Complex n3 = n1.add(n2);
        System.out.println("Number 1 = "+n1);
        System.out.println("Number 2 = "+n2);
        System.out.printf("(%s) + (%s) = %s"
                                  ,n1,n2,n3);

    }
}
```

Inside Complex Class:

```
// Adds two Complex numbers

  public Complex add(Complex num){
       Complex result = new Complex();
       result.real = this.real+num.real;
       result.img = this.img+num.img;
       return result;
    }
```

Return type indicates the class of an object we're returning from the method.

Return an instance of the Complex class



Javadoc

Many of the programmer-defined classes we design are intended to be used by other programmers. It is, therefore, very important to provide meaningful documentation to the client programmers so they can understand how to use our classes correctly.

By adding javadoc comments to the classes we design, we can provide a consistent style of documenting the classes. Once the javadoc comments are added to a class, we can generate HTML files for documentation by using the javadoc command.

Javadoc comments begins with /** and ends with */

Special information such as the authors, parameters, return values, and others are indicated by the @ marker

@param

@author

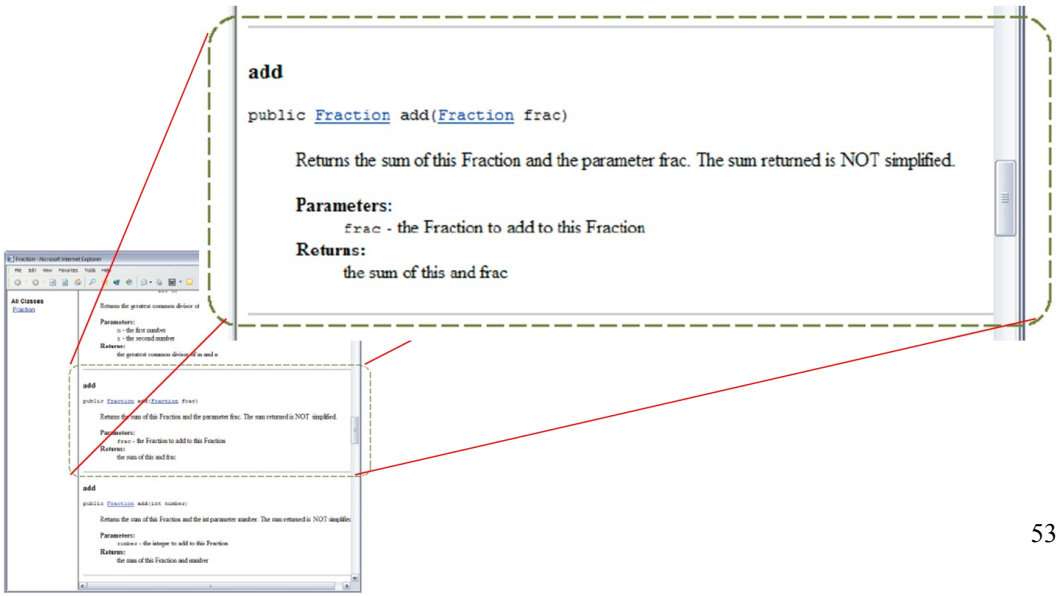
@return

@version

see: http://java.sun.com/j2se/javadoc

```
. . .

/**
* Returns the sum of this Fraction
* and the parameter frac. The sum
* returned is NOT simplified.
*
* @param frac the Fraction to add to this
*             Fraction
*
* @return the sum of this and frac
*/
public Fraction add(Fraction frac) {
   ...
}
. . .
```

this javadoc will produce



Garbage Collection

Garbage collection is the process of automatically finding memory blocks that are no longer being used ("garbage"), and making them available again. In contrast to manual deallocation that is used by many languages, eg C and C++, Java automates this error-prone process and avoids two major problems:

- Dangling references. When memory is deallocated, but not all pointers to it are removed, the pointers are called dangling references -- they point to memory that is no longer valid and which will be reallocated when there is a new memory request, but the pointers will be used as tho they still pointed to the original memory.
- Memory leaks. When there is no longer a way to reach an allocated memory block, but it was never deallocated, this memory will sit there. If this error of not deleting the block occurs many times, eg, in a loop, the program may actually

Exceptions

- An exception represents an error condition that can occur during the normal course of program execution.
- When an exception occurs, or is thrown, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is caught.
- If you do not catch an exception, the exception propagates to the OS, giving a stack trace of the exception

```
inputStr = JOptionPane.showInputDialog(null, "Age:");

try {

    age = Integer.parseInt(inputStr);

} catch (NumberFormatException e){

    JOptionPane.showMessageDialog(null, "'" + inputStr
        +  "' is invalid\n"
        +  "Please enter digits only");
}
```

try

catch

Getting Information on the exception

There are two methods we can call to get information about the thrown exception:

- getMessage
- printStackTrace

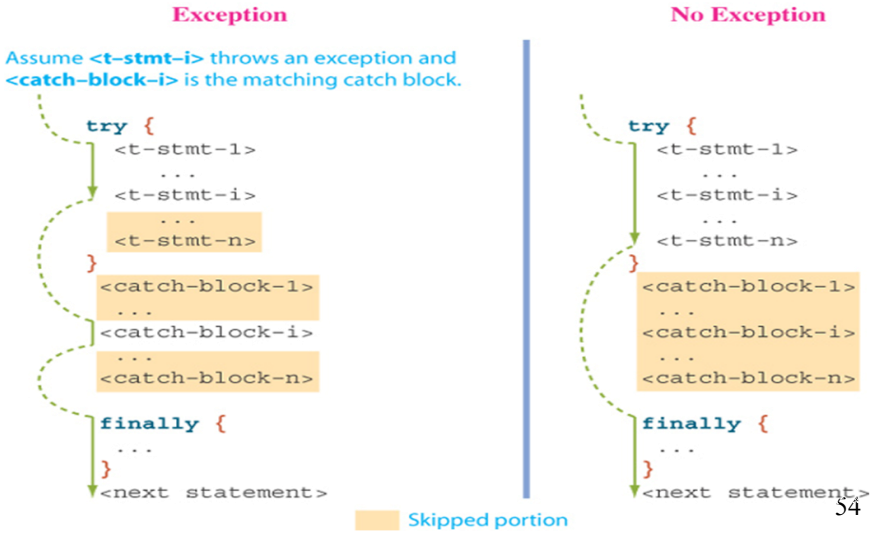
```
try {
    . . .
} catch (NumberFormatException e){

    System.out.println(e.getMessage());
    System.out.println(e.printStackTrace());
}
```

A single try-catch statement can include multiple catch blocks, one for each type of exception.

```
try {
    . . .
    age = Integer.parseInt(inputStr);
    . . .
    val = cal.get(id); //cal is a GregorianCalendar
    . . .
} catch (NumberFormatException e){
    . . .
} catch (ArrayIndexOutOfBoundsException e){
    . . .
}
```

The finally Block

- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions in the finally block.



Propagating Exceptions

Instead of catching a thrown exception by using the try-catch statement, we can propagate the thrown exception back to the caller of our method.

The method header includes the reserved word throws.

```
public int getAge( ) throws NumberFormatException {
    . . .
    int age = Integer.parseInt(inputStr);
    . . .
    return age;
}
```

Throwing Exceptions

We can write a method that throws an exception directly, i.e., this method is the origin of the exception.

Use the throw reserved to create a new instance of the Exception or its subclasses.

The method header includes the reserved word throws.

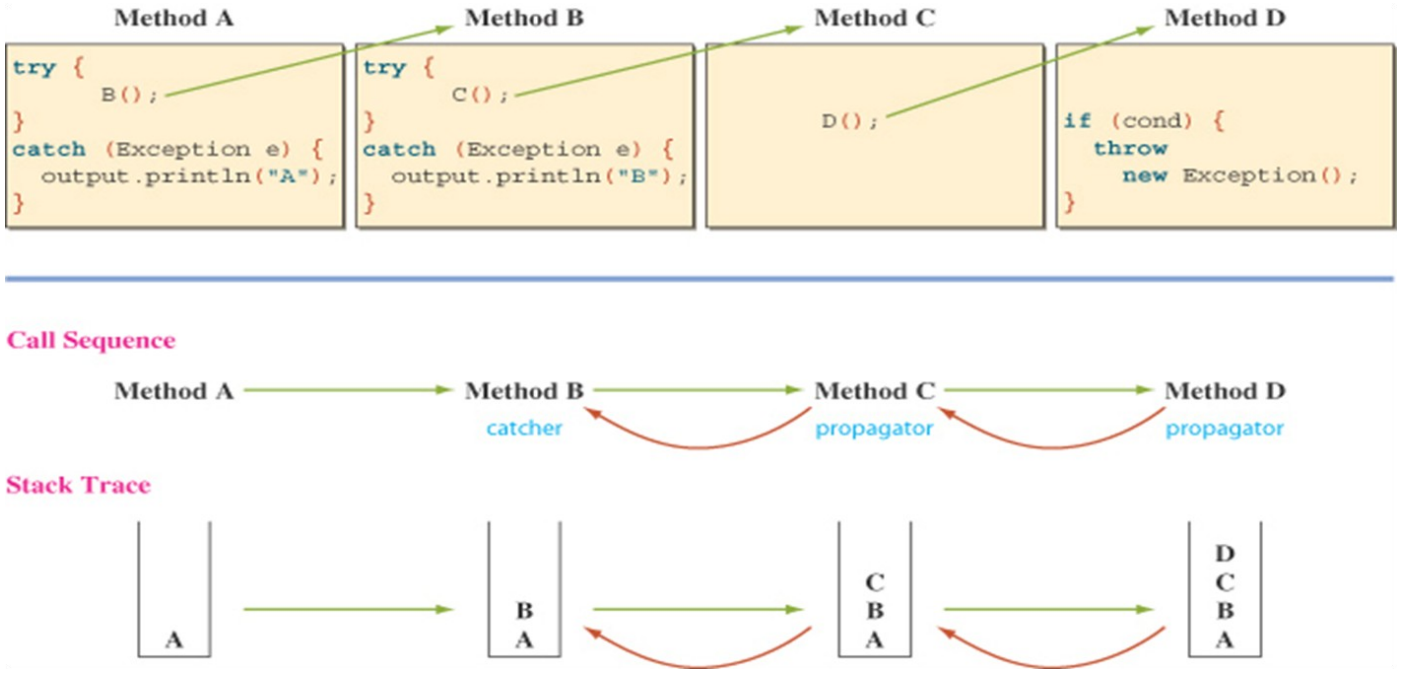
```
public void doWork(int num) throws Exception {
    . . .
    if (num != val) throw new Exception("Invalid val");
    . . .
}
```

Exception Thrower

When a method may throw an exception, either directly or indirectly, we call the method an exception thrower which must be one of two types:

- Exception thrower –catcher. which includes a matching catch block.
- propagator. which does not contain a matching catch block.

A method may be a catcher of one exception and a propagator of another.



Types of Exceptions

There are two types of exceptions:

- Checked.
- Unchecked.

- A checked exception is an exception that is checked at compile time.
- All other exceptions are unchecked, or runtime, exceptions. As the name suggests, they are detected only at runtime.

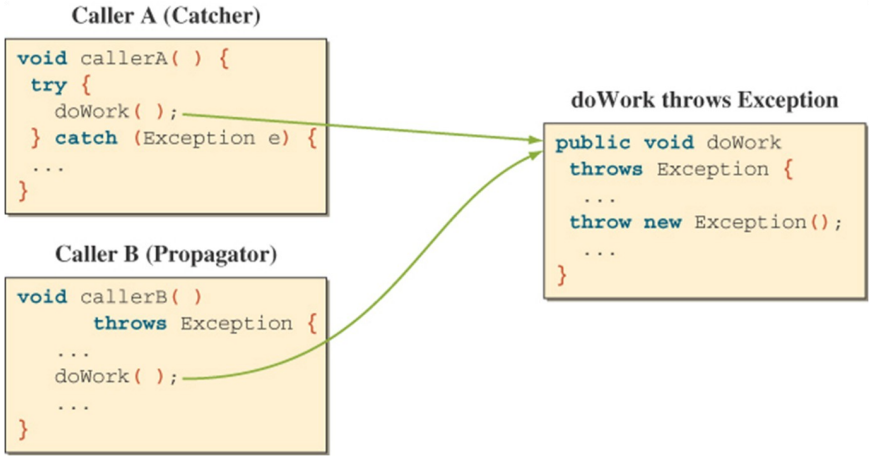
Different Handling Rules

When calling a method that can throw checked exceptions

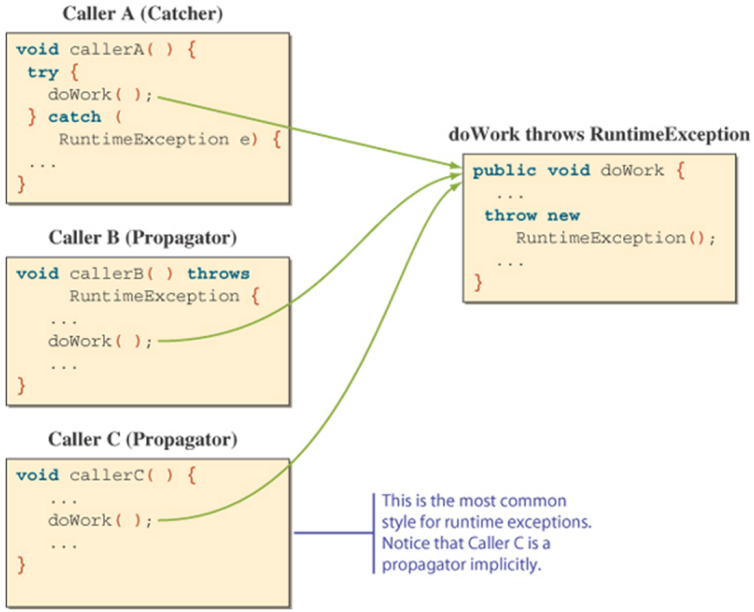
- use the try-catch statement and place the call in the try block, or
- modify the method header to include the appropriate throws clause.

When calling a method that can throw runtime exceptions, it is optional to use the try-catch statement or modify the method header to include a throws clause.

Handling Checked Exceptions



Handling Runtime Exceptions



Programmer-Defined Exceptions

Using the standard exception classes, we can use the getMessage method to retrieve the error message. By defining our own exception class, we can pack more useful information

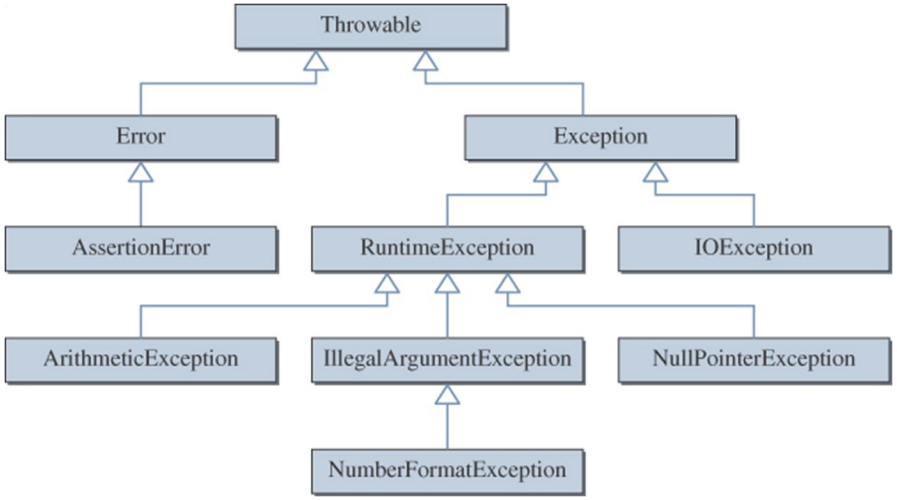
For example, we may define a OutOfStock exception class and include information such as how many items to order

AgeInputException can be defined as a subclass of Exception and includes public methods to access three pieces of information it carries: lower and upper bounds of valid age input and the (invalid) value entered by the user.

```
public class AgeInputException extends Exception {
....
     public String getValue(){
     ...
```

All Children of Throwable

There are over 60 classes in the hierarchy.



Assertions

The syntax for the assert statement is

`assert <boolean expression>;`

where <boolean expression> represents the condition that must be true if the code is working correctly.

If the expression results in false, an AssertionError (a subclass of Error) is thrown.

```
public double deposit(double amount) {

   double oldBalance = balance;

   balance += amount;

   assert balance > oldBalance :
      "Serious Error - balance did not " +
      " increase after deposit";
}
```

Error processing

Error in user input or action

Allow user to recover gracefully.

If the user makes an error in input, give appropriate feedback and allow them to correct the error if possible.

Informative message.

It's important to give specific feedback that allows them to correct their problem. For example a program that produced an "Error in processing" error message. There is no hint as to whether is was a user input error or a bug in the program. Bad.

Debugging error checking - assert

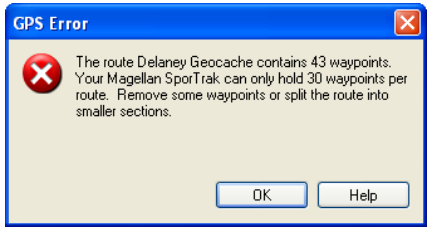
Use assert statements liberally to debug your own code. Assertions are very easy to use and are very helpful during testing. You can leave them in your final code.

Off at runtime.

The disadvantage of assertions is that they are turned off by default during execution. Try to run with them turned on if possible. Some programmers think it's an advantage that they're off at runtime -- there is no execution overhead, and the user won't be faced with potentially puzzling messages.

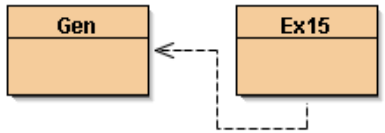
Programmer errors - throw an exception

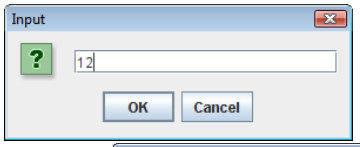
If you write code that is going to be used by other programmers, it is especially important to detect errors and throw an exception.

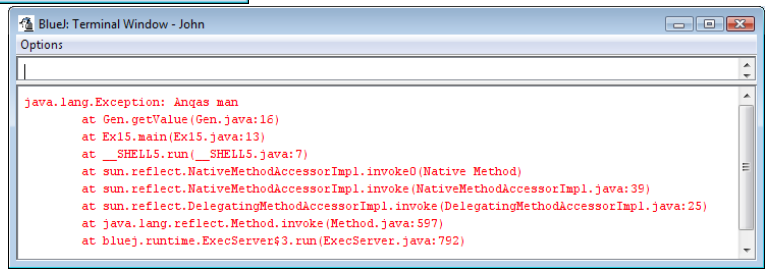


Example 14

```
import javax.swing.*;
/**
 * Gen: generates a new user defined exception
 *
 * @author John Cutajar
 */
public class Gen
{
        public static int getValue() throws Exception{
           String s = JOptionPane.showInputDialog(null);
           int y = Integer.parseInt(s);
           // generate a new exception if number is greater than 11
           if (y>=11) throw new Exception ("Anqas man");
           return y;
          }
}
```







Example 15

```
import javax.swing.*;
/**
 * Example 15:  using commands try, catch and finally.
 * Catches an exception generated by Gen class
 */
public class Ex15
{
   public static void main (String args[]){
      try{
         // try to get an integer from the keyboard
         int z = Gen.getValue();
        }
      // if its not a number
      catch(NumberFormatException err){
        JOptionPane.showMessageDialog(null,"I wanted Integers");
         System.out.println("Error: " + err.getMessage());
         err.printStackTrace(System.out);
      }
      // Or any other exeception
      // Exception Anqas Man gets caught here
      catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

Characters in Java

In Java, single characters are represented using the data type char. Character constants are written as symbols enclosed in single quotes and are stored in a computer memory using some form of encoding.

ASCII, which stands for American Standard Code for Information Interchange, is one of the document coding schemes widely used today.

Java uses Unicode, which includes ASCII, for representing char constants.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
10	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
20	cd4	nak	syn	etb	can	em	sub	esc	fs	gs
30	rs	us	sp	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	}	|	~	del		

For example, character 'O' is 79 (row value 70 + col value 9 = 79).

Unicode Encoding

- The Unicode Worldwide Character Standard (Unicode) supports the interchange, processing, and display of the written texts of diverse languages.
- Java uses the Unicode standard for representing char constants.

```
char ch1 = 'X';

System.out.println(ch1);                    ⟶  X
System.out.println( (int) ch1);             ⟶  88
```

Character Processing

```
char ch1, ch2 = 'X';
```

Declaration and initialization

```
'A' < 'c'
```

This comparison returns true because ASCII value of 'A' is 65 while that of 'c' is 99.

```
System.out.print("ASCII code of character X is " + int) 'X' );

System.out.print("Character with ASCII code 88 is " + (char)88 );
```

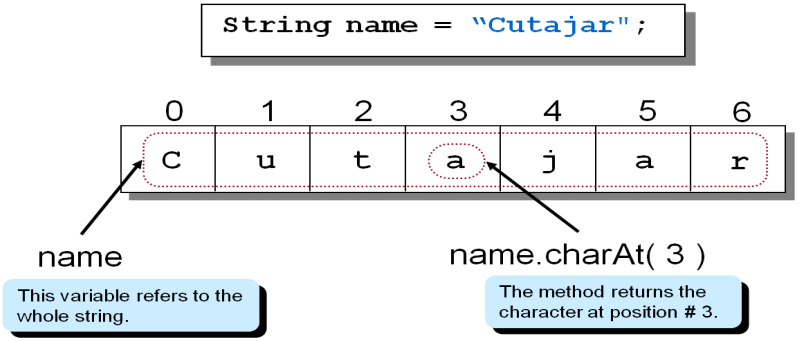
Type conversion between int and char.

Strings

A string is a sequence of characters that is treated as a single value. Instances of the String class are used to represent strings in Java.

Accessing Individual Elements in Strings

Individual characters in a String accessed with the charAt method.



Note: there are far too many methods in class String. See String documentation for further details

Some Other Usefull Methods

compareTo(String anotherString) compareToIgnoreCase(String str)	Compares two strings lexicographically ... ignoring case considerations
endsWith(String suffix)	Tests if this string ends with the specif ied suffix.
equals(String anotherString) equalsIgnoreCase(String anotherString)	Compares this String to another String, ...ignoring case considerations.
indexOf(int ch)	Returns the index within this string of the first occurrence of the specified character.
length()	Returns the length of this string.
matches(String regex)	Tells whether or not this string matches the given regular expression.
startsWith(String prefix)	Tests if this string starts with the specified prefix.
valueOf(long l)	Returns the string representation of the long argument.

Examples

```
char      letter;
String    name          = JOptionPane.showInputDialog(null,"Your name:");
int       numberOfCharacters  = name.length();
int       vowelCount    = 0;

for (int i = 0; i < numberOfCharacters; i++) {
     letter = name.charAt(i);

     if (    letter == 'a' || letter == 'A' ||
             letter == 'e' || letter == 'E' ||
             letter == 'i' || letter == 'I' ||
             letter == 'o' || letter == 'O' ||
             letter == 'u' || letter == 'U'
                                                ) {
          vowelCount++;
     }
}
System.out.print(name + ", your name has " + vowelCount + " vowels");
```

Example: Count the number of vowels in the input string.

```
int       javaCount      = 0;
boolean   repeat         = true;
String    word;

while ( repeat ) {

     word = JOptionPane.showInputDialog(null,"Next word:");

     if ( word.equals("STOP") )      {
          repeat = false;

     } else if ( word.equalsIgnoreCase("Java") ) {
          javaCount++;
     }

}
```

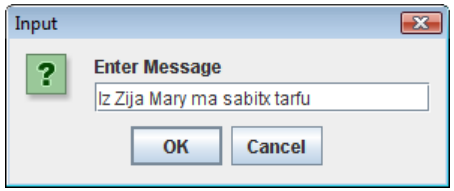
Example: Counting 'Java'.

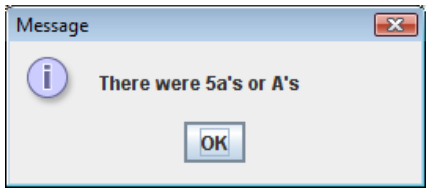
Notice how the comparison is done. We are not using the == operator.

Continue reading words and count how many times the word Java occurs in the input, ignoring the case.

Example 16

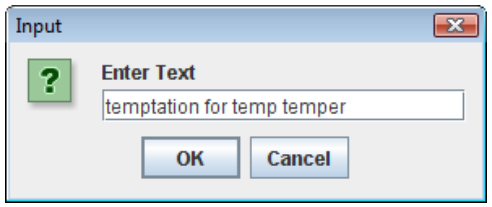
```
import javax.swing.*;
/**
 * This program show the diffenent methods present in String
 * @author John Cutajar
 */
public class Strings
{
   public static void main (String args []){
      String entry ;
      int len, counter, index;
      entry =JOptionPane.showInputDialog(null,"Enter Message ");
      // obtain the length of the string
      len = entry.length();
      //reset counter
      counter = 0;
      // scan the whole string
      for(index = 0; index<len; index++){
         if(entry.charAt(index) == 'a' || entry.charAt(index) == 'A')
            counter++;
      }
      //output the result
      JOptionPane.showMessageDialog(null,"There were "
                                    + counter + " a's or A's");
   }
}
```

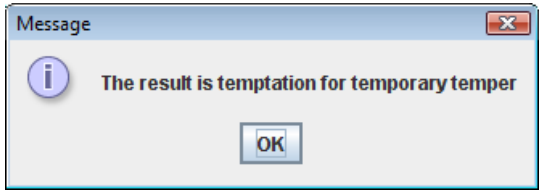




Example 17

```
import javax.swing.*;
/**
 * This program demonstrates the use of the replace
 * @author John Cutajar
 */
public class Replacing {
   public static void main(String args[]){
      String s1,s2;
      s1 =JOptionPane.showInputDialog(null, "Enter Text ");
      //replace all occurrences of temp with temporary
      //(\\b for blank so that temp is replaced temptation no)
      s2 = s1.replaceAll("\\btemp\\b", "temporary");
      //display the result
      JOptionPane.showMessageDialog(null, "The result is " + s2);
   }
}
```



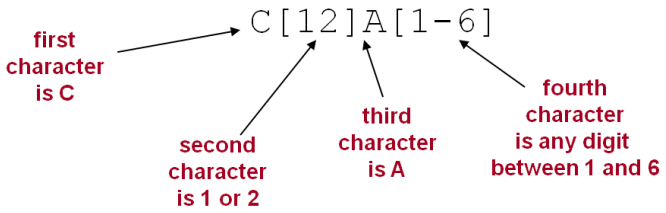


Patterns

Suppose students are assigned a four-character code:

- The first character represents the course
  (C: Computing, I: IT);
- The second digit represents the year (1 or 2);
- The third character represents the level
  (A: Advanced, I Intermediate):
- The fourth digit represents the group number (1 to 6)

The 4-charcter pattern to represent A level Computing students is:



Regular Expressions

The pattern is called a regular expression.

Rules

- The brackets [] represent choices
- The asterisk symbol * means zero or more occurrences.
- The plus symbol + means one or more occurrences.
- The hat symbol ^ means negation.
- The hyphen – means ranges.
- The parentheses ( ) and the vertical bar | mean a range of choices for multiple characters.

Examples

Expression	Description
`[013]`	A single digit 0, 1, or 3.
`[0-9][0-9]`	Any two-digit number from 00 to 99.
`[0-9&&[^4567]]`	A single digit that is 0, 1, 2, 3, 8, or 9.
`[a-z0-9]`	A single character that is either a lowercase letter or a digit.
`[a-zA-z][a-zA-Z0-9_$]*`	A valid Java identifier consisting of alphanumeric characters, underscores, and dollar signs, with the first character being an alphabet.
`[wb](ad|eed)`	Matches `wad`, `weed`, `bad`, and `beed`.
`(AZ|CA|CO)[0-9][0-9]`	Matches `AZxx`,`CAxx`, and `COxx`, where `x` is a single digit.

The replaceAll Method

The replaceAll method replaces all occurrences of a substring that matches a given regular expression with a given replacement string.

Example: Replace all vowels with the symbol @

```
String originalText, modifiedText;

originalText = ...;     //assign string

modifiedText =
          originalText.replaceAll("[aeiou]","@");
```

String Class vs StringBuffer Class

In Java a String object is immutable

This means that once a String object is created, it cannot be changed, such as replacing a character with another character or removing a character

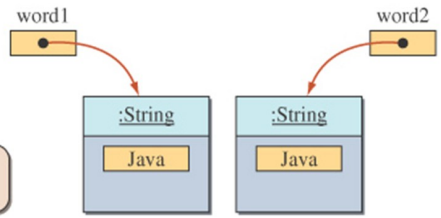
The String methods we have used so far do not change the original string. They created a new string from the original. For example, substring creates a new string from a given string.

The String class is defined in this manner for efficiency reasons.

Effects of Imutability

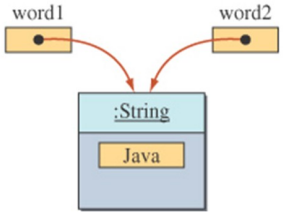
```
String word1, word2;

word1 = new String("Java");

word2 = new String("Java");
```

Whenever the **new** operator is used, there will be a new object.




```
String word1, word2;

word1 = "Java";

word2 = "Java";
```

Literal string constant such as "Java" will always refer to the one object.



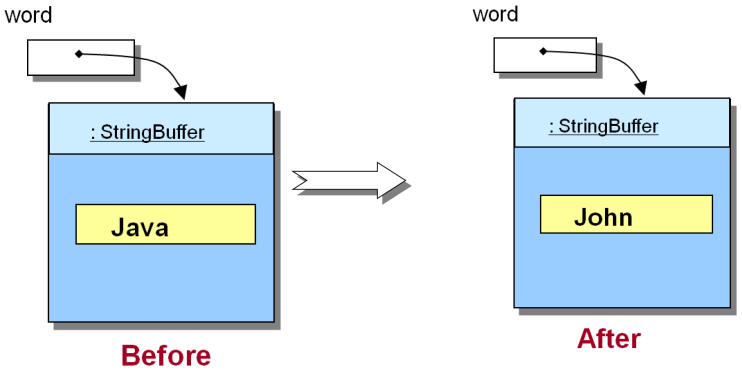
StringBuffer

In many string processing applications, we would like to change the contents of a string. In other words, we want it to be mutable.

similar to StringBuilder class

Manipulating the content of a string, such as replacing a character, appending a string with another string, deleting a portion of a string, and so on, may be accomplished by using the StringBuffer class.

Example : Changing a String Java to John



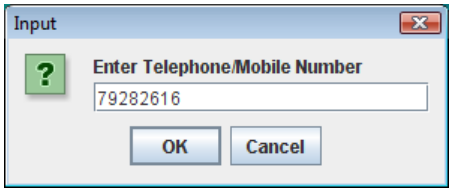

```
StringBuffer word = new StringBuffer("Java");
word.setCharAt(1, 'o');
word.setCharAt(2, 'h');
word.setCharAT(3, 'n');
```

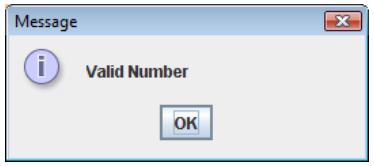
```
char         letter;
String       inSentence    = JOptionPane.showInputDialog(null, "Sentence:");
StringBuffer tempStringBuffer   = new StringBuffer(inSentence);
int          numberOfCharacters = tempStringBuffer.length();

for (int index = 0; index < numberOfCharacters; index++) {

    letter = tempStringBuffer.charAt(index);

    if (letter == 'a' || letter == 'A' || letter == 'e' || letter == 'E' ||
        letter == 'i' || letter == 'I' || letter == 'o' || letter == 'O' ||
        letter == 'u' || letter == 'U'  ) {
        tempStringBuffer.setCharAt(index,'X');
    }
}
JOptionPane.showMessageDialog(null, tempStringBuffer );
```

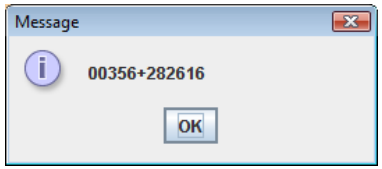
Example: Replace all vowels in the sentence with 'X'.

Example 18

```
import javax.swing.*;
/**
 * checking valid telephone/mobile numbers
 * and demonstrate the use of replaceAll
 * @author John Cutajar
 */
public class Telephone
{
   public static void main (String args []){
     String entry, international ;
      entry =JOptionPane.showInputDialog(null,
                        "Enter Telephone/Mobile Number ");

      if(entry.matches("(21|79|99)[0-9]{6}"))
        JOptionPane.showMessageDialog(null,"Valid Number ");
      else
        JOptionPane.showMessageDialog(null,"Invalid Number");

      //now replace the prefix of the number entered with 00356+
      international = entry.replaceAll("(\\b21|\\b79|\\b99)",
                                                  "00356+");
      JOptionPane.showMessageDialog(null, international);
   }
}
```

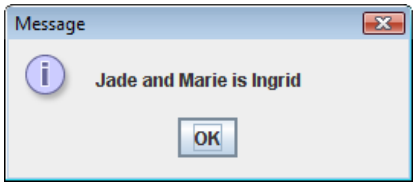






Example 19

```
import javax.swing.*;
/**
 * Demonstrate the use of the string buffers
 * @author John Cutajar
 */
public class StingBuffers
{
   public static void main(String args[]){
      StringBuffer s1 = new StringBuffer("Jake and Marie");

      //replace the char at a particular postition with a new one
      s1.setCharAt(2,'d');

      //add the surname behind the first name
      s1.append(" is Ingid");

      //insert the char v in position 4
      s1.insert(21,'r');

      //display the result
     JOptionPane.showMessageDialog(null, s1);
   }
}
```



Arrays

An array is a collection of data values.

If your program needs to deal with 100 integers, 500 Account objects, 365 real numbers, etc., you will use an array.

In Java, an array is an indexed collection of data values of the same type.

Arrays of Primitive Data Types

- **Array Declaration**

```
<data type> [ ] <variable>        //variation 1
<data type>   <variable>[ ]       //variation 2
```

- **Array Creation**

```
<variable>  = new <data type> [ <size> ]
```

Examples

Variation 1

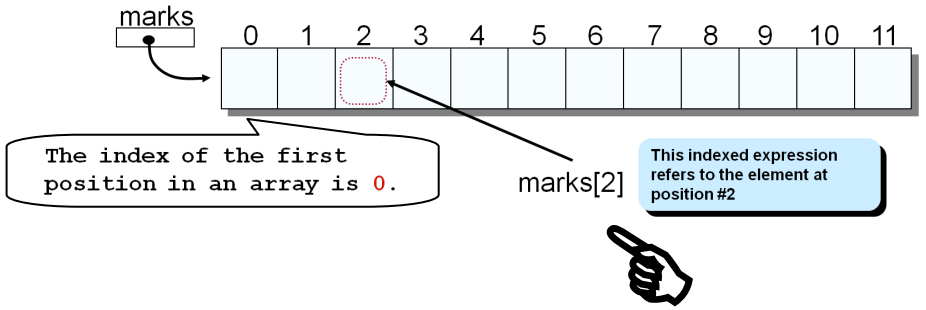
```
float[ ] marks;
marks  = new float[12];
```

Variation 2

```
float marks[ ] ;
marks  = new float[12];
```

An array is like an object!

Accessing Individual Elements




```
//assume rainfall is declared and initialized properly

double[] quarterAverage = new double[4];

for (int i = 0; i < 4; i++) {
    sum = 0;
    for (int j = 0; j < 3; j++) {
                                          //compute the sum of
        sum += rainfall[3*i + j];         //one quarter
    }
    quarterAverage[i] = sum / 3.0;    //Quarter (i+1) average
}
```

Array Initialization

Like other data types, it is possible to declare and initialize an array at the same time.

```
int[] number = { 2, 4, 6, 8 };

double[] samplingData = { 2.443, 8.99, 12.3, 45.009, 18.2,
                          9.00, 3.123, 22.084, 18.08 };

String[] monthName = { "January", "February", "March",
                       "April", "May", "June", "July",
                       "August", "September", "October",
                       "November", "December"  };
```

```
number.length ⟶ 4
samplingData.length ⟶ 9
monthName.length ⟶ 12
```

Variable-size Declaration

In Java, we are not limited to fixed-size array declaration.

The following code prompts the user for the size of an array and declares an array of designated size:

```
int size;
Scanner kb = new Scanner(System.in);
int[] number;
System.out.print("Enter size of array: ");
size= kb.nextInt();

number = new int[size];
```

Arrays of Objects

In Java, in addition to arrays of primitive data types, we can declare arrays of objects. An array of primitive data is a powerful tool, but an array of objects is even more powerful.

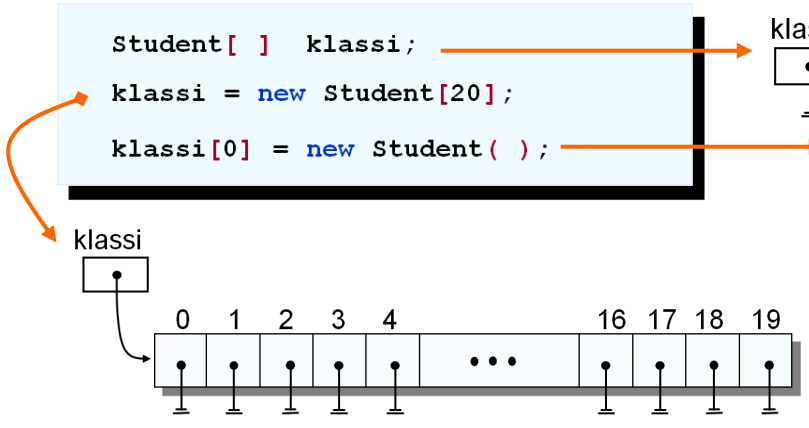
The use of an array of objects allows us to model the application more cleanly and logically.

Example

We will use Student objects to illustrate the use of an array of objects.

```
Student s = new Student( );
s.setName("John Cutajar");
s.setId("217763M");
s.setMark(100);

System.out.println( "Name: " + s.getName() );
System.out.println( "ID : " + s.getId() );
System.out.println( "Mark: " + s.getMark() );
```

The Student class supports the set methods and get methods.



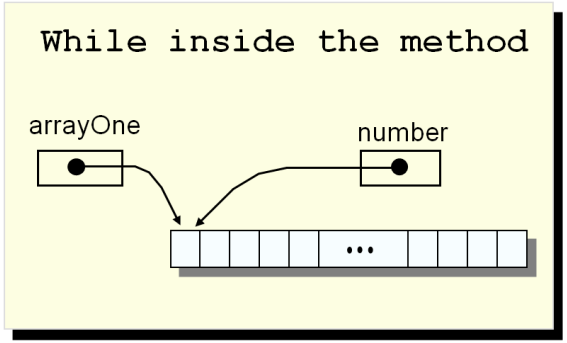
Example: Find the worst and best student.

```
int     worstOne = 0;              //index to the worst student
int     bestOne = 0;               //index to the best student

for (int i = 1; i < klassi.length; i++) {

    if ( klassi[i].getMark() < klassi[worstOne].getMark() ) {
        worstOne          = i;     //found a worse student

    } else if (klassi[i].getMark() > klassi[bestOne].getMark() ) {

        bestOne       = i;         //found an better student
    }
}
//klassi[worstOne] is the worst and klassi[bestOne] is the best
```

Passing Arrays to Methods

```
minOne = searchMinimum(arrayOne);
```

```
public int searchMinimum(float[] number)){
    ...
}
```



Two-Dimensional Arrays

Two-dimensional arrays are useful in representing tabular information.

Multiplication Table

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Declaring and Creating a 2-D Array

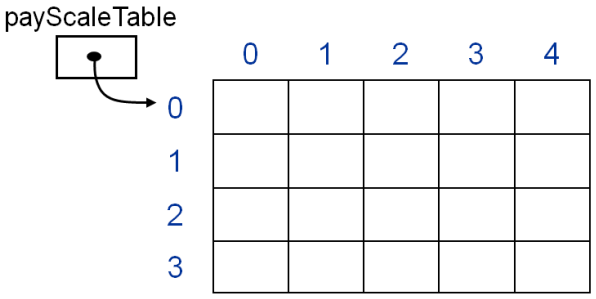
Declaration

```
<data type> [][] <variable>     //variation 1
<data type>   <variable>[][]    //variation 2
```

Creation

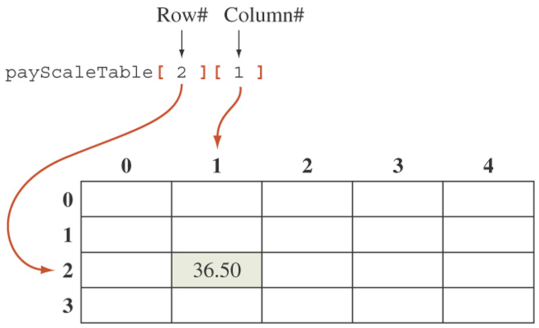
```
<variable>  = new <data type> [ <size1> ][ <size2> ]
```

```
double[][] payScaleTable;
payScaleTable  = new double[4][5];
```



Accessing an Element of a 2-D Array

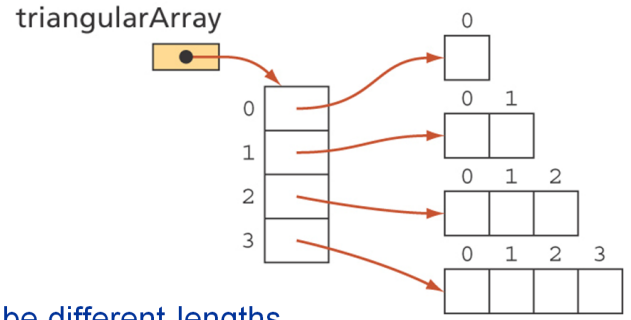
An element in a two-dimensional array is accessed by its row and colmn index.



Example: Find the average of each row.

```
double[ ] average = { 0.0, 0.0, 0.0, 0.0 };

for (int i = 0; i < payScaleTable.length; i++) {

    for (int j = 0; j < payScaleTable[i].length; j++) {

        average[i] += payScaleTable[i][j];
    }

    average[i] = average[i] / payScaleTable[i].length;
}
```

In fact in Java, a two dimensional array is an array of arrays and so the 2D array is not necessarily rectangular

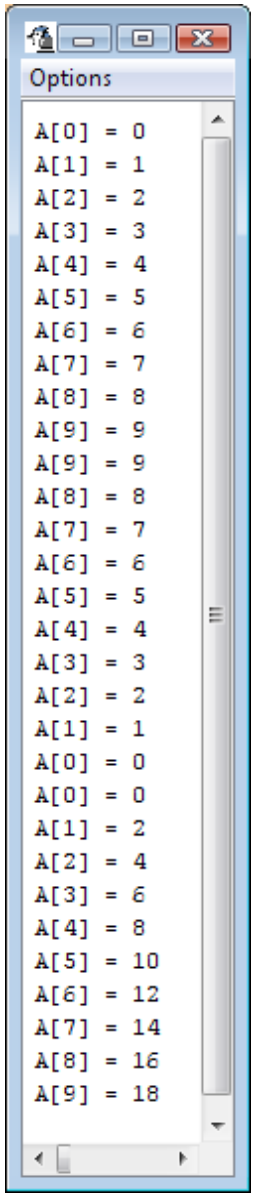


Subarrays may be different lengths.

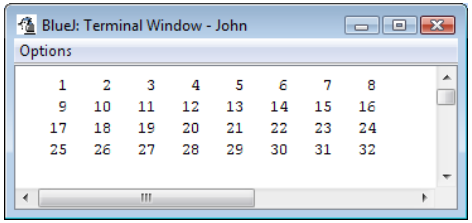
```
triangularArray = new double[4][ ];
for (int i = 0; i < 4; i++)
  triangularArray[i] = new double [i + 1];
```

Example 20

```
/**
 * Demonstrate the use of the Arrays
 * @author John Cutajar
 */
class Arrays{
    public static void main(String args[]){
      //declare an array to hold 10 integer numbers
     int A[] = new int[10];
     // Fill array with the first 10 Numbers starting at 0
     //ascenging order
     for(int i=0; i<10 ; i++){
        A[i] = i;
        System.out.println("A[" + i + "] = " + A[i]);
      }
     //descending order
     for(int i=9; i>=0 ; i--){
        A[i] = i;
        System.out.println("A[" + i + "] = " + A[i]);
      }
       //even only (note i and j used in for)
       for(int i=0, j =0; i<20 ; i++,j += 2){
          A[i] = i;
          System.out.println("A[" + i + "] = " + A[i]);
       }
    }
}
```

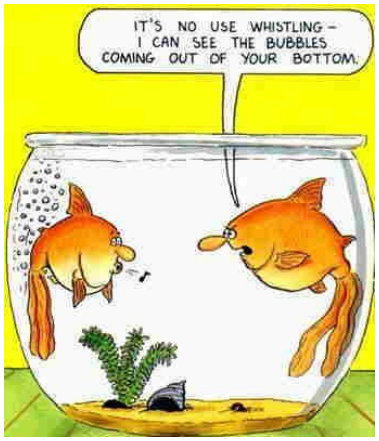


Example 21

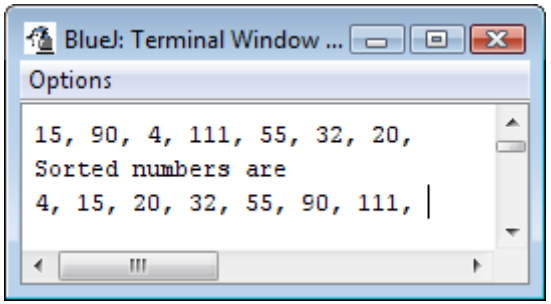



```
/**
 * Example of 2D Arrays
 * @author John Cutajar
 */
public class TwoDArray{
   public static void main(String args[]) {
      //delare the 2D array
      int A[][] = new int [4][8];
      int count = 0;
      //fill the array with values
      for(int i=0; i<4; i++) {
         for(int j=0; j<8; j++)  {
            count++;
            A[i][j] = count;
         }
      }
      //now display the values in table form
      for(int i=0; i<4; i++) {
         for(int j=0; j<8; j++)  {
             System.out.printf("%5d", (A[i][j]));
         }
         // display a fresh line
         System.out.println();
      }
   }
```

Example 22



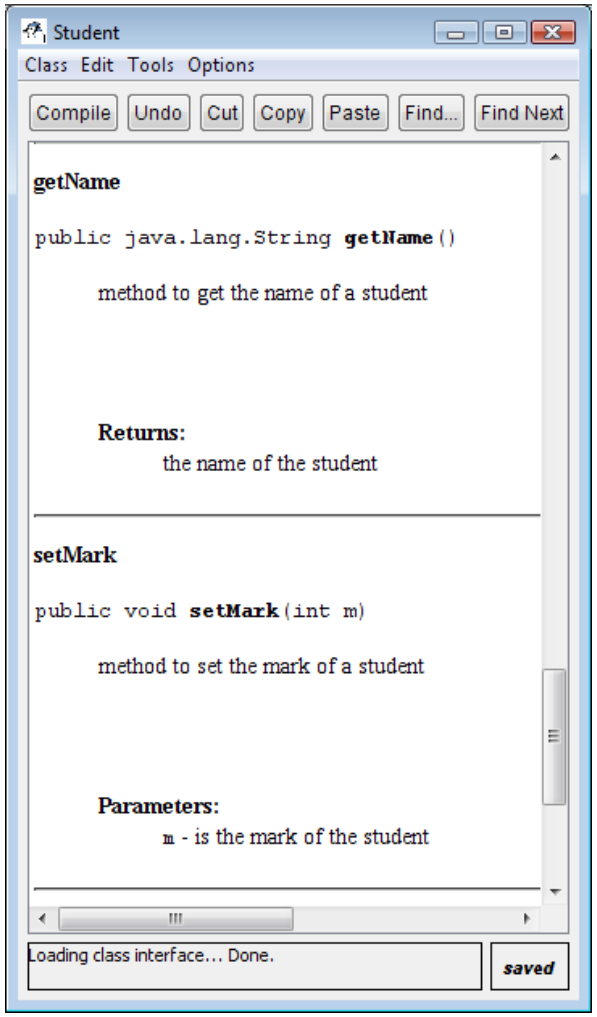

```
/**
 * Implementing the Bubble sort algorithm
 * @author John Cutajar
 */
class BubbleSort
{
        public static void main(String args[])
        {
                //declare and define the number of elements
                final int SIZE = 7;

                //declare the integer array
                int A[] = {15,90,4,111,55,32,20};
                int temp = 0;

                //now place some numbers in it
                //display the unsorted data

                for(int i=0; i<SIZE; i++)
                {
                        System.out.print(A[i]+", ");
                }
                System.out.println("");

                //This is the Bubble Sort
                for(int i=0; i<SIZE; i++)
                {
                        for(int j=SIZE-2; j>=0; j--)
                        {
                                if(A[j] > A[j+1])
                                {
                                        temp = A[j];
                                        A[j] = A[j+1];
                                        A[j+1] = temp;
                                }
                        }
                }

                //Printing the Sorted list
                System.out.println("Sorted numbers are ");
                for(int i=0; i<SIZE; i++)
                {
                        System.out.print(A[i]+", ");
                }
        }
}
```



Student Class

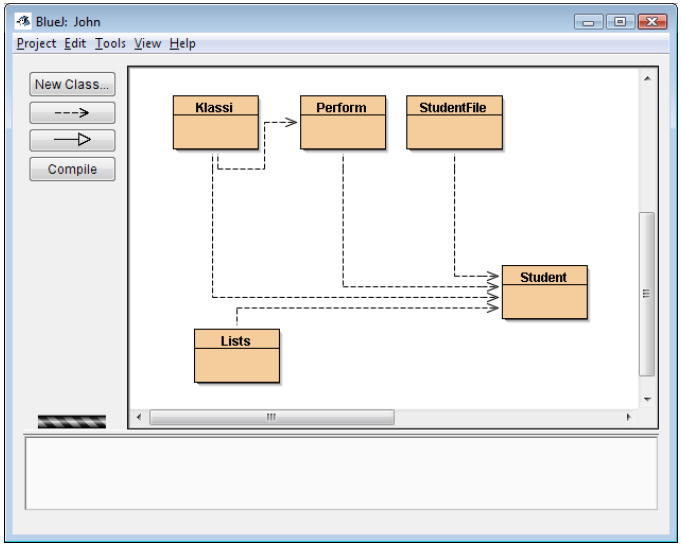
```
import java.io.*;
/**
 * Demo Class to be used in arrays and files
 * @author John Cutajar
 */
public class Student implements Serializable{
   //Data Members
   private String name;
   private int mark;
   // Parameterless Constructor
   public Student() {
      name = " ";
      mark = 0;
   }
   // Parametrized Constructor
   public Student(String s, int m) {
      name = s;
      mark = m;
   }
    //Member methods
    /**
    * method to set the name of a student
    * @param s is the name of the student
    */
   public void setName(String s){
      name = s;
   }
   /**
    * method to get the name of a student
    * @returns the name of the student
    */
   public String getName() {
      return name;
   }
    /**
    * method to set the mark of a student
    * @param m is the mark of the student
    */
   public void setMark(int m){
      mark = m;
   }
   /**
    * method to get the name of a student
    * @returns the mark of the student
    */
   public int getMark() {
      return mark;
   }
}
```

Demo of javadoc



Performance Class

```
/**
 * Class for static methods for a "class"
 * @author John Cutajar
 */
public class Perform{
   //Get the best student in the class
   public static Student getBest(Student[] myClass){
      int bestindex = 0;
      // scan the class and remember where you found the best
      for(int i=0; i < myClass.length; i++){
         if(myClass[i].getMark() > myClass[bestindex].getMark())
         bestindex = i;
      }
      return myClass[bestindex];
   }
   //Get the worst student of the class
   public static Student getWorst(Student[] myClass){
      int worstindex = 0;
      //Scan the class and remember the worst
      for(int i=0; i < myClass.length; i++){
         if(myClass[i].getMark() < myClass[worstindex].getMark())
         worstindex = i;
      }
      return myClass[worstindex];
   }
   // Calculate the average
   public static float getAverage(Student[] myClass){
      float average;
      //Remember to reset the accumulator variable !!!
      int total = 0;
      //Accumulate Total by adding class marks
      for(int i = 0; i < myClass.length; i++){
         total += myClass[i].getMark();
      }
      //Calculate average
      average = total/myClass.length;
      return average;
   }
}
```



Example 23

```
import java.util.*;
/**
 * Demo for an array of Students
 * @author John Cutajar
 * Uses objects Students and static methods in Perform
 */
public class Klassi{
   public static void main (String args []) {
        Scanner kb = new Scanner(System.in);
        //Get the number of students in the class
        System.out.print(" please enter size of class:   ");
        int size = kb.nextInt();
        // Declare an array of Student Objects
        Student[] klassi = new Student [size];
        // Change delimiter to enter name and surname at @1go
        kb.useDelimiter("\n");
        // Enter students details
        for(int i=0; i<size;i++){
           System.out.print("Enter Student's name " );
           String isem = kb.next();
           System.out.print(" Enter corrisponding mark ");
           int marka = kb.nextInt();
           Student s = new Student(isem,marka);
           klassi[i]=s;
         }
        // get the best student
        Student best = Perform.getBest(klassi);
        System.out.println("Best Student is: " + best.getName()
                + " with "  + best.getMark() + " marks");
        // get the worst student
        Student worst = Perform.getWorst(klassi);
        System.out.println("Worst Student is: " +
            worst.getName()+" with  "+worst.getMark()  + " marks ");
        //get class average
        float average = Perform.getAverage(klassi);
        System.out.println("The Average is: " + average);
      }
}
```

Lists and Maps

The java.util standard package contains different types of classes for maintaining a collection of objects.

These classes are collectively referred to as the Java Collection Framework (JCF).

JCF includes classes that maintain collections of objects as sets, lists, or maps.

Java Interface

A Java interface defines only the behavior of objects

- It includes only public methods with no method bodies.
- It does not include any data members except public constants
- No instances of a Java interface can be created

JCF Lists

JCF includes the List interface that supports methods to maintain a collection of objects as a linear list

$$L = (l0, l1, l2, \ldots, lN)$$

We can add to, remove from, and retrieve objects in a given list.

A list does not have a set limit to the number of objects we can add to it.

Here are some of the 25 list methods:

`boolean add(Object o)`	**Adds an object o to the list**
`void clear()`	**Clears this list, i.e., make the list empty**
`Object get(int idx)`	**Returns the element at position idx**
`boolean remove(int idx)`	**Removes the element at position idx**
`int size()`	**Returns the number of elements in the list**

Using Lists

To use a list in a program, we must create an instance of a class that implements the List interface.

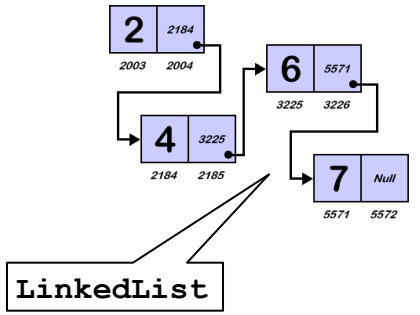
Two classes that implement the List interface:

- ArrayList
- LinkedList

The ArrayList class uses an array to manage data.

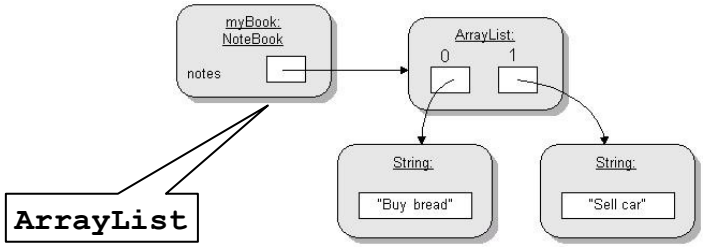
The LinkedList class uses a technique called linked-node representation.

Example: manipulating a list:



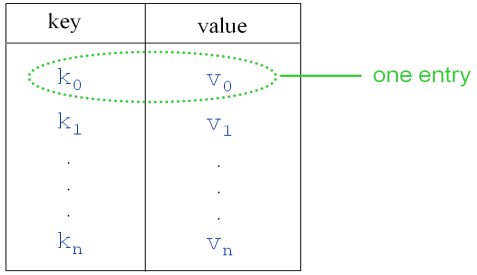

```
import java.util.*;

List    klassi;
Student newStudent;

klassi = new ArrayList();

newStudent = new Student("John","217763M",100);
klassi.add(newStudent);
newStudent = new Student("Maria","456763M",99);
klassi.add(newStudent);

Student s = (Student) klassi.get(1);
```

There are various collections in Java—another popular one, very similar to the ArrayList is the Vector.



JCF Maps

JCF includes the Map interface that supports methods to maintain a collection of objects (key, value) pairs called map entries.



Map Methods

Here are some of the 14 list methods:

`void clear()`	Clears this list, i.e., make the map empty
`boolean containsKey( Object key )`	Returns true if the map contains an entry with a given key
`Object put(Object key, Object value)`	Adds the given (key, value) entry to the map
`boolean remove(Object key)`	Removes the entry with the given key from the map
`int size()`	Returns the number of elements in the map

Using Maps

To use a map in a program, we must create an instance of a class that implements the Map interface.

Two classes that implement the Map interface:

- HashMap
- TreeMap

Example: Manipulating a map:

```
import java.util.*;

Map     catalog;
catalog = new TreeMap( );

catalog.put("CS101", "Intro Java Programming");
catalog.put("CS301", "Database Design");
catalog.put("CS413", "Software Design for Mobile Devices");

if (catalog.containsKey("CS101")) {
    System.out.println("We teach Java this semester");
} else {
    System.out.println("No Java courses this semester");
}
```

Iterators

Iterators let you process each element of a Collection. Iterators are a nice generic way to go through all the elements of a Collection no matter how it is organised. Iterator is an Interface implemented a different way for every Collection.

```
import java.util.*;
...
ArrayList a = new ArrayList( 100 );
a.add( "broccoli" );
a.add( "cauliflower" );

Iterator iter = a.iterator();
while (iter.hasNext())
    {
    String value = (String)iter.next();
    // display each vegetable
    System.out.println( value );
    }
```

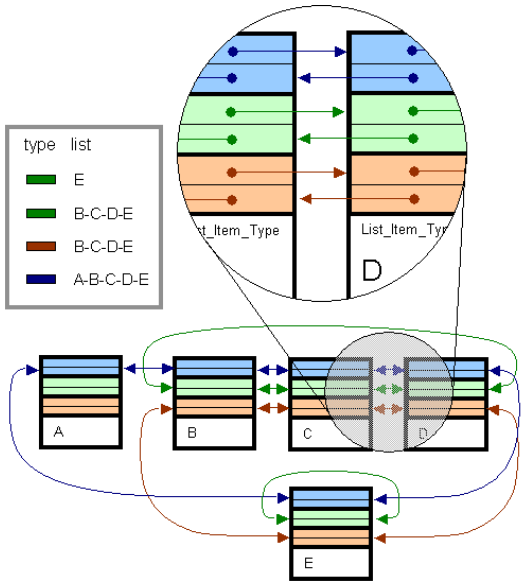
for-each loop

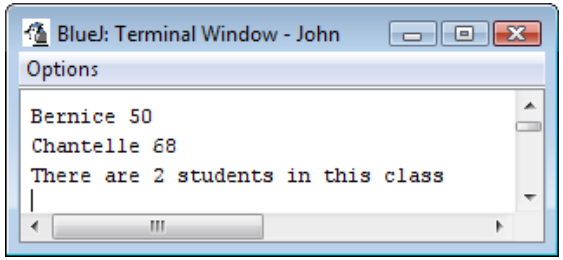
The basic for loop was extended in Java 5 to make iteration over arrays and other collections more convenient. This newer for statement is called the enhanced for or for-each

```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (double d : ar) { // d gets successively each value in ar.
          sum += d;
}
```

Example 24

```
import java.util.*;
/**
 * using lists instead of arrays for a "class"
 * @author John Cutajar
 */

public class Lists{
   public static void main (String args[]){

      List <Student> klassi;
      Student student;

      klassi = new ArrayList <Student> ();

      // add three students to the list
      student = new Student ("Bernice", 50);
      klassi.add(student);

      student = new Student ("Claudia", 38);
      klassi.add(student);

      student = new Student ("Chantelle", 68);
      klassi.add(student);

      // remove the second student
      klassi.remove(1);

      //Use of iterators mekes it more elegant to scan
      Iterator ptr;
      ptr = klassi.iterator();

      while(ptr.hasNext()){
         student = (Student)ptr.next();
         System.out.println(student.getName() + " " +
student.getMark());
      }
      System.out.println("There are " + klassi.size() + "
students in this class");
   }
}
```





File Input and Output

The File Class

To operate on a file, we must first create a File object (from java.io).

```
File inFile = new File("sample.dat");
```

Opens the file sample.dat in the current directory.

```
File inFile = new File ("C:/SamplePrograms/test.dat");
```

Opens the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the full pathname.

Some File Methods

```
if ( inFile.isFile() ) {
```

To see if inFile is associated to a file or not. If false, it is a directory.

```
if ( inFile.exists( ) ) {
```

To see if inFile is associated to a real file correctly.

```
File directory = new File("C:/JavaProjects");

String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
    System.out.println(filename[i]);
}
```

List the name of all files in the directory C:\JavaProjects

The JFileChooser Class

A javax.swing.JFileChooser object allows the user to select a file from a specific directory.

```
JFileChooser chooser = new JFileChooser("D:/JavaProjects");
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
    JOptionPane.showMessageDialog(null, "Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
    JOptionPane.showMessageDialog(null, "Cancel is clicked");
}
File currentDirectory = chooser.getCurrentDirectory();
File selectedFile  = chooser.getSelectedFile();
```

Low-Level File I/O

To read data from or write data to a file, we must create one of the Java stream objects and attach it to the file.

A stream is a sequence of data items, usually 8-bit bytes.

Java has two types of streams: an input stream and an output stream.

An input stream has a source form which the data items come, and an output stream has a destination to which the data items are going.

Streams for Low-Level File I/O

FileOutputStream and FileInputStream are two stream objects that facilitate file access.

- FileOutputStream allows us to output a sequence of bytes; values of data type byte.
- FileInputStream allows us to read in an array of bytes.

```
File  outFile   = new File("sample1.data");

FileOutputStream outStream = new FileOutputStream( outFile );

//data to save
byte[] byteArray = {10, 20, 30, 40,50, 60, 70, 80};

//write data to the stream
outStream.write( byteArray );

//close the stream
outStream.close();
```

```
File inFile  = new File("sample1.data");
FileInputStream inStream = new FileInputStream(inFile);

//set up an array to read data in
int    fileSize  = (int)inFile.length();
byte[] byteArray = new byte[fileSize];

//read data in and display them
inStream.read(byteArray);
for (int i = 0; i < fileSize; i++) {
       System.out.println(byteArray[i]);
}

//close the stream
inStream.close();
```

Streams for High-Level File I/O

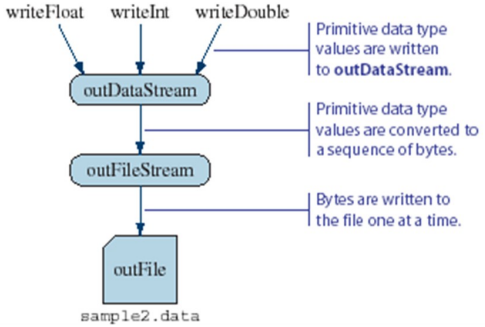
FileOutputStream and DataOutputStream are used to output primitive data values whilst FileInputStream and DataInputStream are used to input primitive data values

To read the data back correctly, we must know the order of the data stored and their data types

Setting up DataOutputStream

A standard sequence to set up a DataOutputStream object:

```
File             outFile       = new File("sample2.data");
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```



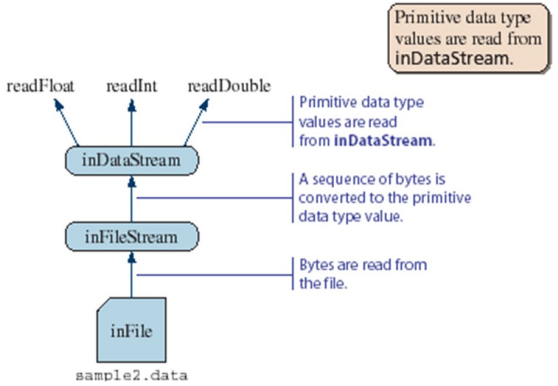
Example

```
import java.io.*;
class Ch12TestDataOutputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up outDataStream

        //write values of primitive data types to the stream
        outDataStream.writeInt(987654321);
        outDataStream.writeLong(11111111L);
        outDataStream.writeFloat(22222222F);
        outDataStream.writeDouble(3333333D);
        outDataStream.writeChar('A');
        outDataStream.writeBoolean(true);

        //output done, so close the stream
        outDataStream.close();
    }
}
```

Setting up DataInputStream

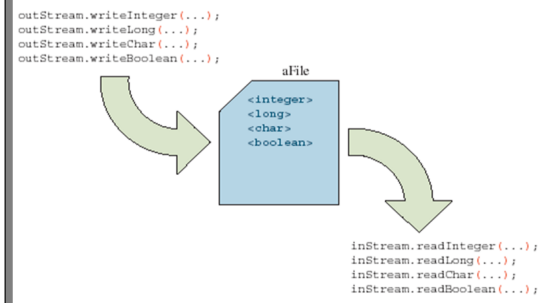
A standard sequence to set up a DataInputStream object:

```
File            inFile       = new File("sample2.data");
FileInputStream inFileStream = new FileInputStream(inFile);
DataInputStream inDataStream = new DataInputStream(inFileStream);
```



Example

```
import java.io.*;
class Ch12TestDataInputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up inDataStream

        //read values back from the stream and display them
        System.out.println(inDataStream.readInt());
        System.out.println(inDataStream.readLong());
        System.out.println(inDataStream.readFloat());
        System.out.println(inDataStream.readDouble());
        System.out.println(inDataStream.readChar());
        System.out.println(inDataStream.readBoolean());

        //input done, so close the stream
        inDataStream.close();
    }
}
```



The order of write and read operations must match in order to read the stored primitive data back correctly.

Textfile Input and Output

Instead of storing primitive data values as binary data in a file, we can convert and store them as a string data.

This allows us to view the file content using any text editor

To output data as a string to file, we use a PrintWriter object

To input data from a textfile, we use FileReader and BufferedReader classes

From Java 5.0 (SDK 1.5), we can also use the Scanner class for inputting textfiles

Example

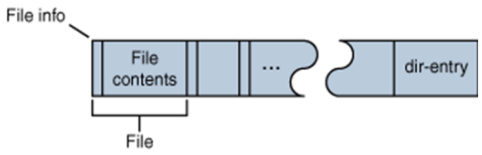
```
import java.io.*;
class TestBufferedReader {

    public static void main (String[] args) throws IOException {

        //set up file and stream
        File inFile = new File("sample3.data");
        FileReader fileReader = new FileReader(inFile);
        BufferedReader bufReader = new BufferedReader(fileReader);
        String str;

        str = bufReader.readLine();
        int i = Integer.parseInt(str);

        //similar process for other data types

        bufReader.close();
    }
}
```

```
import java.io.*;
class TestPrintWriter {
    public static void main (String[] args) throws IOException {

        //set up file and stream
        File outFile = new File("sample3.data");
        FileOutputStream outFileStream
                        = new FileOutputStream(outFile);
        PrintWriter outStream = new PrintWriter(outFileStream);

        //write values of primitive data types to the stream
        outStream.println(987654321);
        outStream.println("Hello, world.");
        outStream.println(true);

        //output done, so close the stream
        outStream.close();
    }
}
```

Textfile Input with Scanner

```
import java.io.*;

class TestScanner {

    public static void main (String[] args) throws IOException {

        //open the Scanner
        Scanner scanner = new Scanner(new File("sample3.data"));

        //get integer
        int i = scanner.nextInt();

        //similar process for other data types

        scanner.close();
    }
}
```

Random Access Files

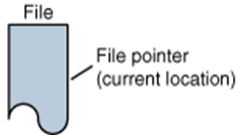


Random access files permit nonsequential, or random, access to a file's contents.

The following code creates a RandomAccessFile named cutajar.txt and opens it for both reading and writing:

```
new RandomAccessFile("cutajar.txt", "rw");
```

RandomAccessFile supports the notion of a file pointer. The file pointer indicates the current location in the file. When the file is first created, the file pointer is set to 0, indicating the beginning of the file. Calls to the read and write methods adjust the file pointer by the number of bytes read or written.



In addition to the normal file I/O methods that implicitly move the file pointer when the operation occurs, RandomAccessFile contains three methods for explicitly manipulating the file pointer.

`int skipBytes(int)` — forward by the specified number of bytes

`void seek(long)` — Positions the file pointer just before specified byte

`long getFilePointer()` — Returns location of the file pointer

Object File I/O

It is possible to store objects just as easily as you store primitive data values. We use ObjectOutputStream and ObjectInputStream to save to and load objects from a file.

To save objects from a given class, the class declaration must include the phrase implements Serializable. For example,

```
class Person implements Serializable {
        . . .
}
```

objects must implement this interface to be able to be stored in a file

```
File                outFile   = new File("objects.data");
FileOutputStream    outStream = new FileOutputStream(outFile);
ObjectOutputStream  outObject = new ObjectOutputStream(outStream);
…
newTeacher   = new Teacher("Cutajar","Computing",44);
newStudent   = new Student("Zammit","23233G",45);

outObject.writeObject(newTeacher);
outObject.writeObject(newStudent);
```

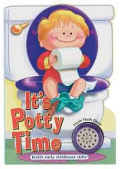
Could save objects from the different classes.

```
File                inFile   = new File("objects.data");
FileInputStream     inStream = new FileInputStream(inFile);
ObjectInputStream   inObject = new ObjectInputStream(inStream);

Teacher newTeacher = (Teacher) inObject.readObject( );
Bank    newStudent = (Student) inObject.readObject( );
```

Must read in the correct order and type cast to the correct object type.

flush

flush makes sure everything you have written so far is committed to the hard disk, and the expanded file length is also committed to the disk directory, with the updated lastModified timestamp committed too. If you crash the system later, you know at least that much is guaranteed to be there on disk waiting for you when you reboot.



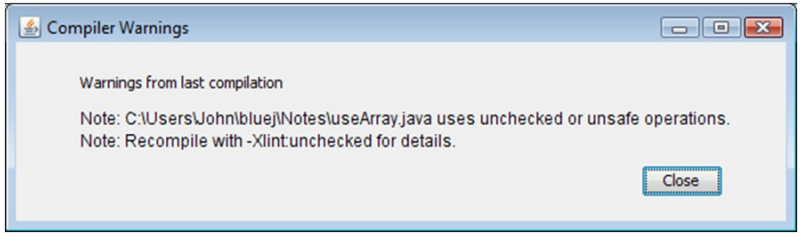
Saving and Loading Arrays

Instead of processing array elements individually, it is possible to save and load the whole array at once.

```
Student[] klassi = new Student[ N ];
            //assume N already has a value

//build the people array
. . .
//save the array
outObject.writeObject (klassi);
```

```
//read the array
Student[ ] klassi = (Student[]) inObject.readObject( );
```

Some Compiler Warning on Arrays

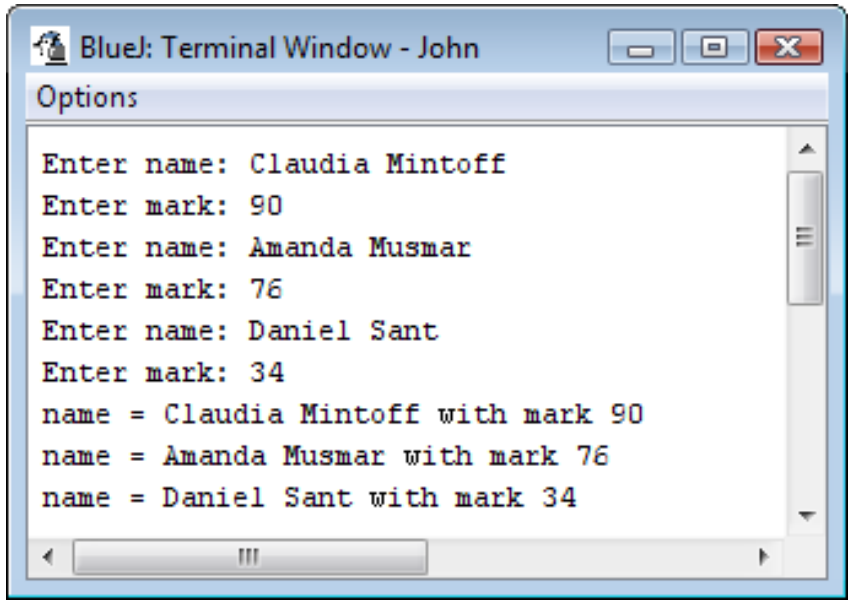


We have seen that in array, and saving arrays to file, the array can store any object. Nevertheless teh compiler gives a warning when compiling untyped arrays. This could safely be ignored if you know exactly what you are doing, or else type the array by means of the <>

```
ArrayList <String> a = new ArrayList<String>( 100 );
```

Example 24

```
import java.io.*;
import java.util.*;
/**
 * Example to demonstrate file of objects
 * @author John Cutajar
 */
public class StudentFile{
   public static void main (String args []) throws Exception{
      // Create an output file klassi.dat in the current working directory
      File myfile = new File ("klassi.dat");
      FileOutputStream myStream = new FileOutputStream(myfile);
      ObjectOutputStream myObject = new ObjectOutputStream(myStream);

      Scanner kb = new Scanner(System.in);
      kb.useDelimiter("\n");

      //read three students and store them in the file
      for(int i = 0; i < 3; i++){
         System.out.print("Enter name: ");
         String name = kb.next();
         System.out.print("Enter mark: ");
         int mark = kb.nextInt();
         Student s = new Student(name,mark);
         myObject.writeObject(s);
      }
      //very important to close the file to flush changes
      myObject.close();

      // open an input file for reading
      FileInputStream yourStream = new FileInputStream(myfile);
      ObjectInputStream yourObject = new ObjectInputStream(yourStream);

      for(int i = 0; i < 3; i++){
         Student st = (Student)yourObject.readObject();
         System.out.println("name = " + st.getName() + " with mark " +
                            st.getMark());
      }
      yourObject.close();
   }
}
```



Inheritance and Polymorphism

Case Study:

- Suppose we want to implement a model of class roster that contains both first year and second year students at the junior college.
- Each student's record will contain his or her name, three assessment marks, and the final exam grade for first years and two assessments and project mark for second years.
- The formula for determining if the student passed the year is different for first and second year students.

Modelling Two Types of Students

There are two ways to design the classes to model first and second year students.

- We can define two unrelated classes, one for first and one for second years.
- We can model the two kinds of students by using classes that are related in an inheritance hierarchy.
- Two classes are unrelated if they are not connected in an inheritance relationship

Classes for the Class Roster

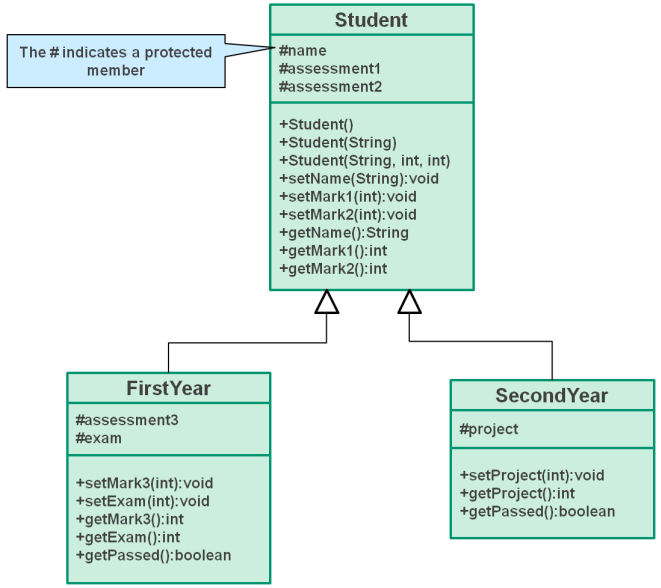
For the Class Roster example, we design three classes:

- Student
- FirstYear
- SecondYear

The Student class will incorporate behavior and data common to both FirstYear and SecondYear objects.

The FirstYear class and the SecondYear class will each contain behaviours and data specific to their respective objects.

Inheritance Hierarchy



Defining a Subclass

To define a subclass of another class, we declare the subclass with the reserved word extends

```
class FirstYear extends Student {
    . . .
}
```

```
class SecondYear extends Student {
    . . .
}
```

The Protected Modifier

The modifier protected makes a data member or method visible and accessible to the instances of the class and the descendant classes.

Polymorphism

Polymorphism allows a single variable to refer to objects from different subclasses in the same inheritance hierarchy

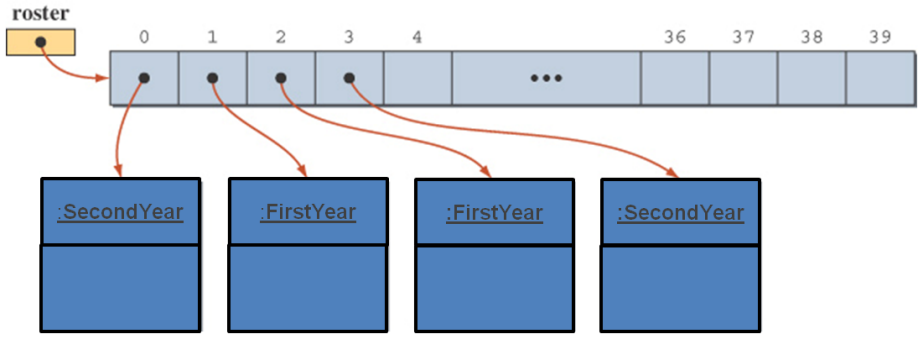
For example, if Cat and Dog are subclasses of Pet, then the following statements are valid:

```
Pet myPet;

myPet = new Dog();
. . .
myPet = new Cat();
```

Creating the roster Array

We can maintain our class roster using an array, combining objects from the Student, FirstYear, and SecondYear classes.

```
Student roster = new Student[40];
. . .
roster[0] = new SecondYear();
roster[1] = new FirstYear();
roster[2] = new FirstYear();
. . .
```



Polymorphic Message

```
// In FirstYear Class
public boolean getPassed(){
    return( ((int) assessment1+assessment2+assessment3+(exam*0.7F))>45);
}
```

if SeconfdYear → `roster[1].getPassed();` ← if FirstYear

```
// In SecondYear Class
public boolean getPassed(){
    return( ((int) assessment1+assessment2+(project*0.8F))>45);
}
```

The instanceof Operator

The instanceof operator can help us learn the class of an object.

The following code counts the number of first year students.

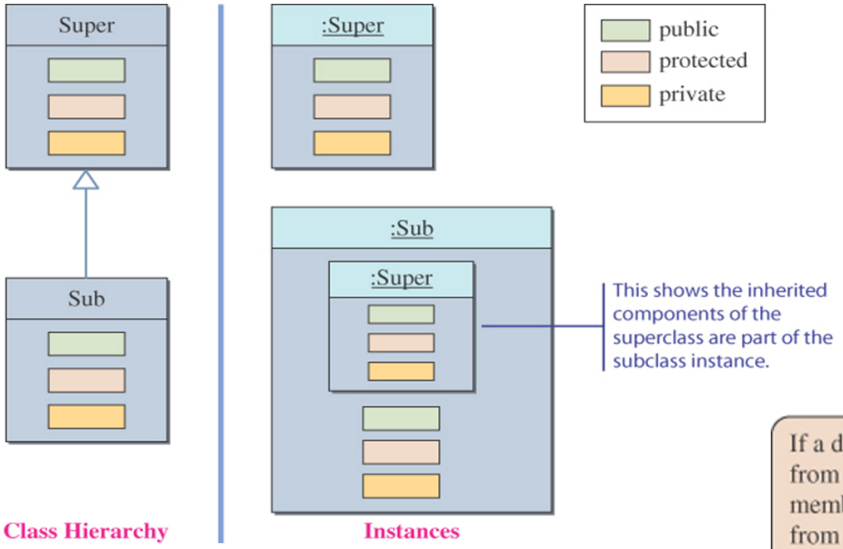
```
int firstYearCount = 0;
for (int i = 0; i < roster.length(); i++) {
    if ( roster[i] instanceof FirstYear ) {
        firstYearCount++;
    }
}
```

Method Overriding

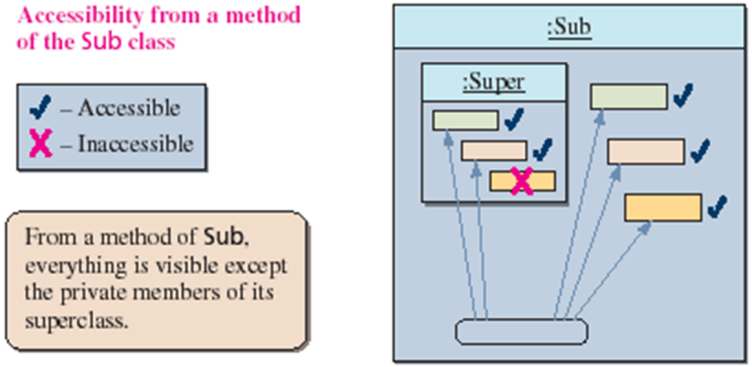
Overriding is to provide a replacement method is a new class for one in the superclass. The superclass too will use your new method in place of its own when dealing with objects of your type, though it will continue to use its own method for objects purely of its own type. In the above example if getPassed was already defined in student, it will get used only on object of class Student, but not for FirstYear or SecondYear classes.

Inheritance and Member Accessibility

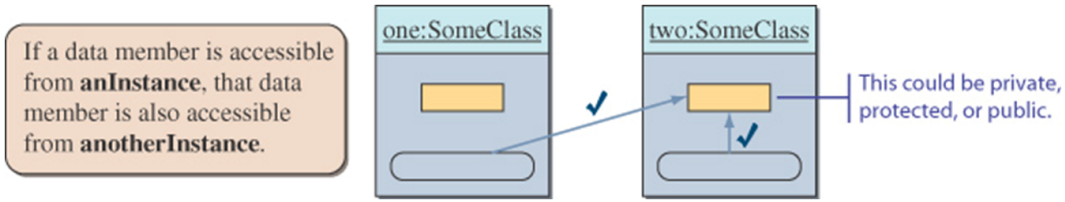
We use the following visual representation of inheritance to illustrate data member accessibility



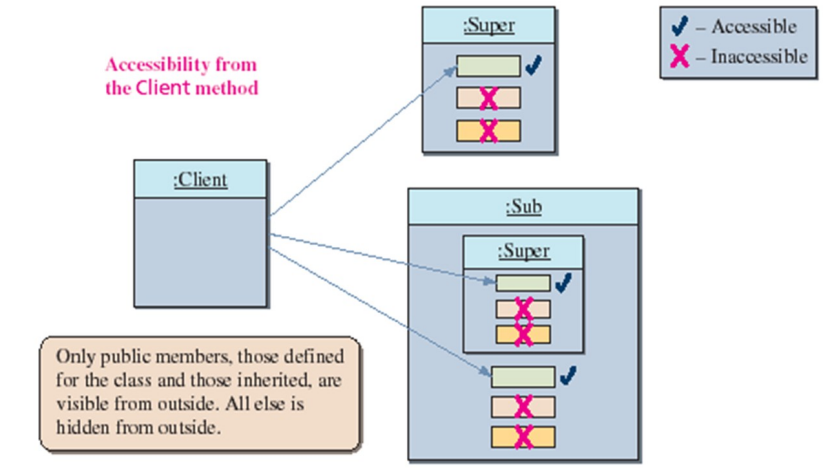
Accessibility of Super from Sub



Accessibility from Another Instance



The Effect of Three Visibility Modifiers



Inheritance and Constructors

Unlike members of a superclass, constructors of a superclass are not inherited by its subclasses.

You must define a constructor for a class or use the default constructor added by the compiler.

The statement

`super();` calls the superclass's constructor.

If the class declaration does not explicitly designate the superclass with the extends clause, then the class's superclass is the Object class. super is also used to call an overridden method of a superclass .

```
super.getPassed();
```

Abstract Superclasses and Abstract Methods

When we define a superclass, we often do not need to create any instances of the superclass. In this case we must define the class differently as an abstract class.

Definitions

An abstract class is a class

- defined with the modifier abstract OR
- that contains an abstract method OR
- that does not provide an implementation of an inherited abstract method

An abstract method is a method with the keyword abstract, and it ends with a semicolon instead of a method body. Private methods and static methods may not be declared abstract, and no instances can be created from an abstract class.

Case 1: Student Must Be FirstYear or SecondYear

- If a student must be either a FirstYear or a SecondYear student, we only need instances of FirstYear or SecondYear.
- Therefore, we must define the Student class so that no instances may be created of it.

Case 2 Student Does Not Have to be FirstYear or SecondYear.

- In this case, we may design the Student class in one of two ways.
- We can make the Student class instantiable.
- We can leave the Student class abstract and add a third subclass, OtherStudent, to handle a student who does not fall into the FirstYear or SecondYear categories.

Abtract methods are heads without a body

Which Approach to Use

The best approach depends on the particular situation.

When considering design options, we can ask ourselves which approach allows easier modification and extension.

The Java Interface

A Java interface includes only constants and abstract methods.

An abstract method has only the method header, or prototype. There is no method body. You cannot create an instance of a Java interface.

A Java interface specifies a behavior.

A class implements an interface by providing the method body to the abstract methods stated in the interface.

Any class can implement the interface.

Inheritance versus Interface

The Java interface is used to share common behaviour (only method headers) among the instances of different classes.

Inheritance is used to share common code (including both data members and methods) among the instances of related classes.

In your program designs, remember to use the Java interface to share common behaviour. Use inheritance to share common code.

If an entity A is a specialized form of another entity B, then model them by using inheritance. Declare A as a subclass of B.

Inheritance of an abstract class

If a class inherits an abstract class it must either provide an implementation for that method, or declare that method again as abstract. This is useful to force the programmer to provide a method which cannot be implemented at the time of definition of the superclass

FirstYear Class

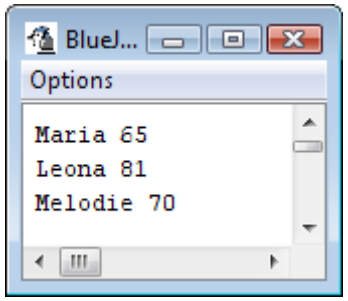
```
/**
 * Class describing First Year students.
 * @author John Cutajar
 */
public class FirstYear extends Student{
   // assuming Superclass Student mark contains total
assesment
   private int examMark;

   public FirstYear(String name, int assessment, int exam)
   {
      super(name,assessment);
      examMark = exam;
   }

   public boolean hasPassed()
   {
      // the first year has passed if total marks >= 45
      return(this.getMark()>=45);
   }

   //Overriding Student getMark()
   public int getMark(){
      return(super.getMark() + (int) (examMark*0.7));
   }
}
```

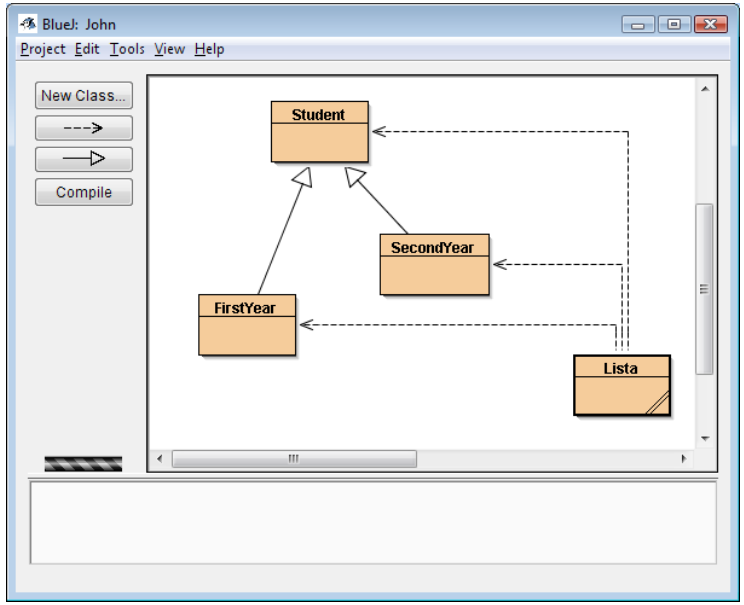
SecondYear Class

```
/**
 * Class describing Second Year students.
 * @author John Cutajar
 */
public class SecondYear extends Student{
   // assuming Superclass Student mark contains total
assesment
   private int projectMark;

   public SecondYear(String name, int assessment, int project)
   {
      super(name,assessment);
      projectMark = project;
   }

   public boolean hasPassed()
   {
      // the first year has passed if total marks >= 45
      return(this.getMark()>=45);
   }

   //Overriding Student getMark()
   public int getMark(){
      return(super.getMark() + (int) (projectMark*0.8));
   }
}
```

Example 25

```
import java.util.*;
/**
 * using lists instead of arrays for a klassi
 * @author John Cutajar
 */

public class Lista{
   public static void main (String args[]){

      List <Student> klassi;
      Student student;

      klassi = new ArrayList <Student> ();

      // add three students to the list

      student = new SecondYear("Maria", 38, 34);
      klassi.add(student);

      student = new FirstYear ("Leona",34,68);
      klassi.add(student);

      // This would be illegal if Student was abstract
      student = new Student ("Melodie",70);
      klassi.add(student);

      Iterator ptr;
      ptr = klassi.iterator();

      while(ptr.hasNext()){
         student = (Student)ptr.next();
         System.out.println(student.getName() + " " +
                                          student.getMark());
      }
   }
}
```





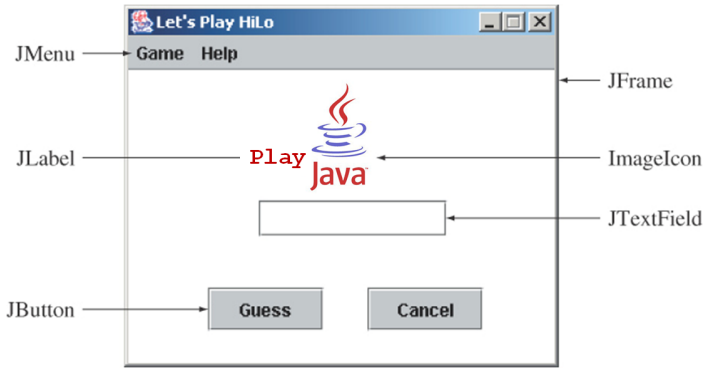
Graphical User Interface

In Java, GUI-based programs are implemented by using classes from the javax.swing and java.awt (older) packages.

The Swing classes provide greater compatibility across different operating systems. They are fully implemented in Java, and behave the same on different operating systems.

Sample GUI Objects

Various GUI objects from the javax.swing package.



Plain JFrame

The JFrame class contains rudimentary functionalities to support features found in any frame window.

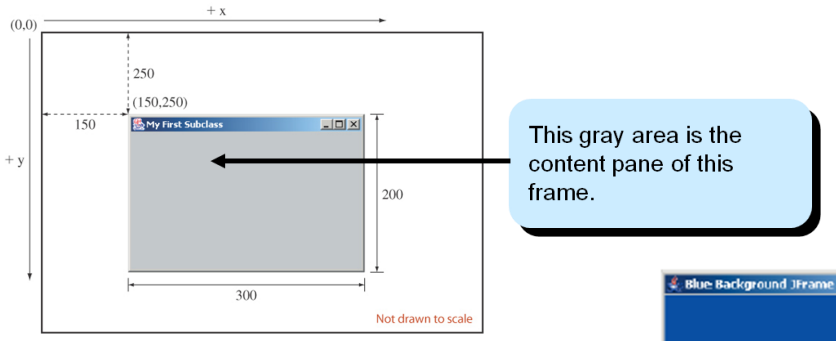
```
import javax.swing.*;
class DefaultJFrame {
    public static void main( String[] args ) {
        JFrame defaultJFrame;
        defaultJFrame = new JFrame();
        defaultJFrame.setVisible(true);
    }
}
```

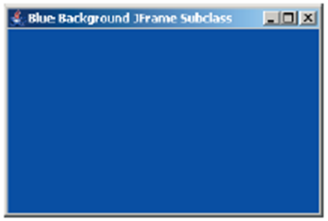
Subclassing JFrame

To create a customized frame window, we define a subclass of the JFrame class.

```
import javax.swing.*;

class JFrameSubclass1 extends JFrame {
  public Ch14JFrameSubclass1( ) {
        setTitle      ( "My First Subclass" );
        setSize       ( 300, 200 );
        setLocation   ( 150, 250 );

        setDefaultCloseOperation( EXIT_ON_CLOSE );
    }
}
```

notice no object - we're in it





The Content Pane of a Frame

The content pane is where we put GUI objects such as buttons, labels, scroll bars, and others.

We access the content pane by calling the frame's getContentPane() method.

```
Container contentPane = getContentPane();
contentPane.setBackground(Color.BLUE);
```

Placing GUI Objects on a Frame

There are two ways to put GUI objects on the content pane of a frame:

- Use a layout manager
  - FlowLayout
  - BorderLayout
  - GridLayout
- Use absolute positioning null layout manager

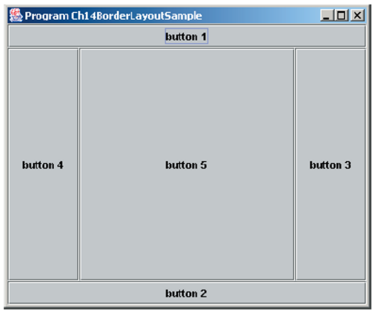
The layout manager determines how the GUI components are added to the container (such as the content pane of a frame)

BorderLayout

This layout manager divides the container into five regions: center, north, south, east, and west.

Not all regions have to be occupied.

After the frame is resized.

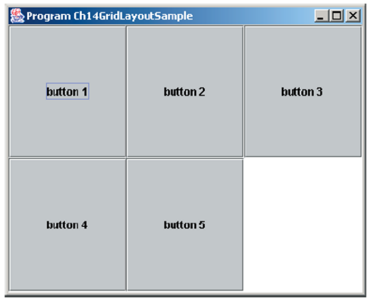


GridLayout

This layout manager places GUI components on equal-size N by M grids.

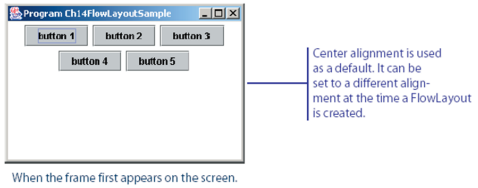
Components are placed in top-to-bottom, left-to-right order.

After the frame is resized.



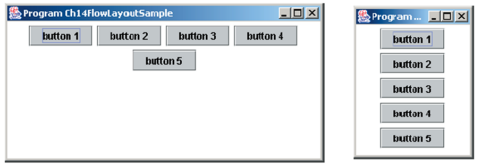
FlowLayout

In using this layout, GUI components are placed in left-to-right order.



When the frame first appears on the screen.

When the component does not fit on the same line, left-to-right placement continues on the next line.



After the frame's width is widened and shortened.

As a default, components on each line are centered.

When the frame containing the component is resized, the placement of components is adjusted accordingly.

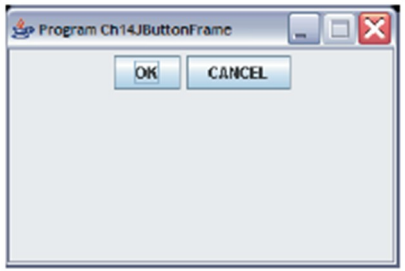
Nested Panels

A better approach is to use multiple panels, placing panels inside other panels.

Example : placing a button with FlowLayout.

A JButton object a GUI component that represents a pushbutton.

```
contentPane.setLayout(new FlowLayout());
okButton = new JButton("OK");
cancelButton = new JButton("CANCEL");
contentPane.add(okButton);
contentPane.add(cancelButton);
```



Event Handling

An action involving a GUI object, such as clicking a button, is called an event. and the mechanism to process events is called event handling.

The event-handling model of Java is based on the concept known as the delegation-based event model.

With this model, event handling is implemented by two types of objects:

- event source objects
- event listener objects

Event Source Objects

An event source is a GUI object where an event occurs. We say an event source generates events.

Buttons, text boxes, list boxes, and menus are common event sources in GUI-based applications.

Although possible, we do not, under normal circumstances, define our own event sources when writing GUI-based applications.

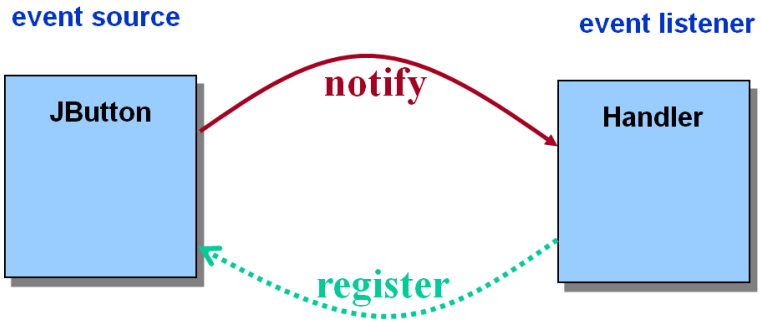
Event Listener Objects

An event listener object is an object that includes a method that gets executed in response to the generated events.

A listener must be associated, or registered, to a source, so it can be notified when the source generates events.

Connecting Source and Listener

A listener must be registered to a event source. Once registered, it will get notified when the event source generates events.



Event Types

Registration and notification are specific to event types

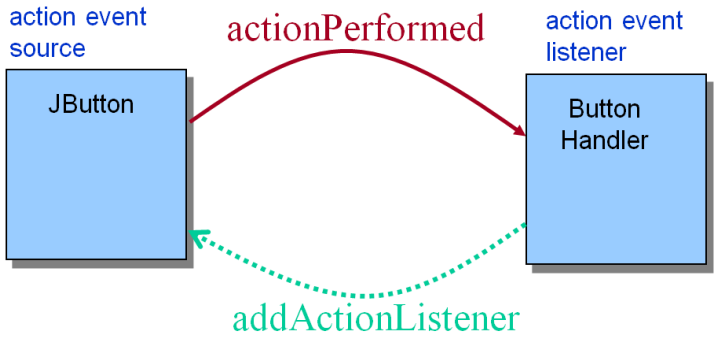
- Mouse listener handles mouse events
- Item listener handles item selection events
- and so forth

Among the different types of events, the action event is the most common.

- Clicking on a button generates an action event
- Selecting a menu item generates an action event
- and so forth

Action events are generated by action event sources and handled by action event listeners.

Handling Action Events




```
JButton button = new JButton("OK");
ButtonHandler handler = new ButtonHandler( );
button.addActionListener(handler);
```

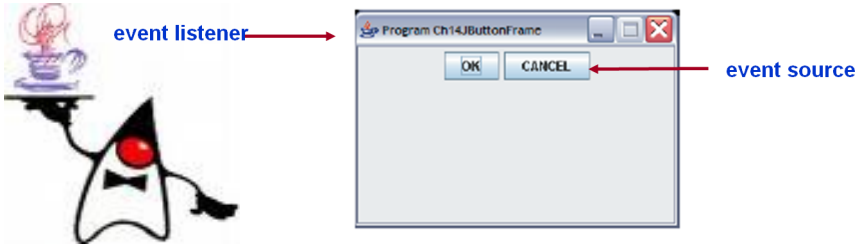
ActionListener Interface

When we call the addActionListener method of an event source, we must pass an instance of a class that implements the ActionListener interface.

The ActionListener interface includes one method named actionPerformed.

A class that implements the ActionListener interface must therefore provide the method body of actionPerformed.

Since actionPerformed is the method that will be called when an action event is generated, this is the place where we put a code we want to be executed in response to the generated events.



Example: The ButtonHandler Class

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class ButtonHandler implements ActionListener {
    . . .
    public void actionPerformed(ActionEvent event) {
        JButton clickedButton = (JButton) event.getSource();

        JRootPane rootPane = clickedButton.getRootPane( );
        Frame     frame    = (JFrame) rootPane.getParent();

        frame.setTitle("You clicked " + clickedButton.getText());
    }
}
```

Container as Event Listener

Instead of defining a separate event listener such as ButtonHandler, it is much more common to have an object that contains the event sources be a listener.

Example: we make this frame a listener of the action events of the buttons it contains.

```
. . .
class ButtonFrameHandler extends JFrame
                         implements ActionListener {
   . . .
   public void actionPerformed(ActionEvent event) {
        JButton clickedButton
                    = (JButton) event.getSource();

        String  buttonText = clickedButton.getText();

        setTitle("You clicked " + buttonText);
    }
}
```

GUI Classes for Handling Text

The Swing GUI classes JLabel, JTextField, and JTextArea deal with text.

- A JLabel object displays uneditable text (or image).
- A JTextField object allows the user to enter a single line of text.
- A JTextArea object allows the user to enter multiple lines of text. It can also be used for displaying multiple lines of uneditable text.

JTextField

We use a JTextField object to accept a single line to text from a user. An action event is generated when the user presses the ENTER key.

The getText method of JTextField is used to retrieve the text that the user entered.

```
JTextField input = new JTextField( );
input.addActionListener(eventListener);
contentPane.add(input);
```

JLabel

We use a JLabel object to display a label. A label can be a text or an image. When creating an image label, we pass ImageIcon object instead of a string.

```
JLabel textLabel = new JLabel("Please enter your name");
contentPane.add(textLabel);
JLabel imgLabel = new JLabel(new ImageIcon("java.gif"));
contentPane.add(imgLabel);
```

JTextArea

We use a JTextArea object to display or allow the user to enter multiple lines of text.

The setText method assigns the text to a JTextArea, replacing the current content.

The append method appends the text to the current text.

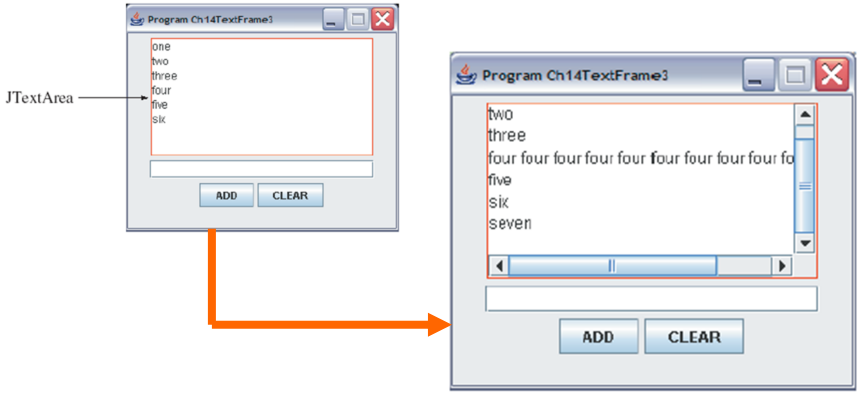
```
JTextArea textArea = new JTextArea( );
. . .
textArea.setText("Min gera mexa,\n");
textArea.append("min ma geriex,");
textArea.append("intesa");
```

```
Min gera mexa,
min ma geriex, intesa
```

JTextArea

Adding Scroll Bars to JTextArea

```
JTextArea   textArea   = new JTextArea();
. . .
JScrollPane scrollText = new JScrollPane(textArea);
. . .
contentPane.add(scrollText);
```

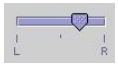


Other Common GUI Components

- JCheckBox → 

- JRadioButton → 

- JComboBox → 

- JSlider → 

- JList → 

Menus

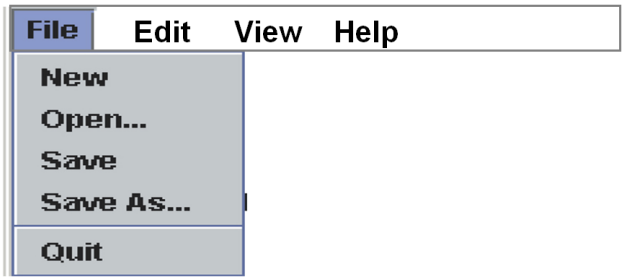
The javax.swing package contains three menu-related classes: JMenuBar, JMenu, and JMenuItem.

JMenuBar is a bar where the menus are placed. There is one menu bar per frame.

JMenu (such as File or Edit) is a group of menu choices. JMenuBar may include many JMenu objects.

JMenuItem (such as Copy, Cut, or Paste) is an individual menu choice in a JMenu object.

Only the JMenuItem objects generate events.



Sequence for Creating Menus

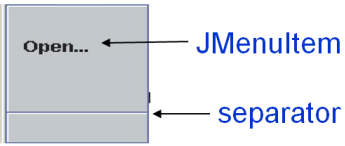
1.Create a JMenuBar object and attach it to a frame.

JMenuBar 

2.Create a JMenu object.

JMenu 

3.Create JMenuItem objects and add them to the JMenu object.



4.Attach the JMenu object to the JMenuBar object.

Handling Mouse Events

Mouse events include such user interactions as

- moving the mouse
- dragging the mouse (moving the mouse while the mouse button is being pressed)
- clicking the mouse buttons.

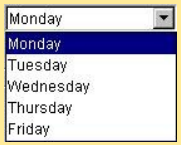
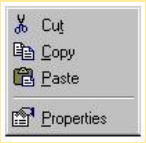
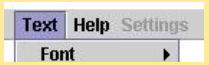
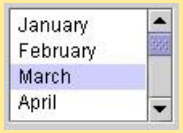
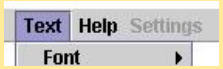
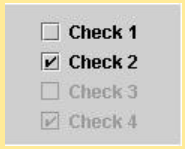
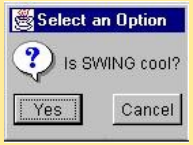
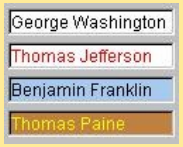
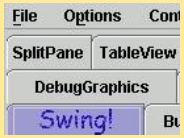
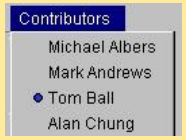
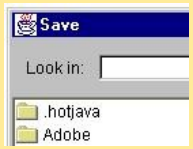
The MouseListener interface handles mouse button

mouseClicked, mouseEntered, mouseExited, mousePressed, and mouseReleased

The MouseMotionListener interface handles mouse movement

- mouseDragged and mouseMoved.

Some swing components

Borders	JComboBox	JLabel	JPopupMenu	JTree	JMenu
					
JButton	**ImageIcon**	**JList**	**JRadioButton**	**JSplitPane**	**JMenuBar**
					
JCheckBox	**JDialog**	**JTextField**	**JColorChooser**	**JTabbedPane**	**JScrollBar**
					
CheckBoxMenuItem	**FileChooser**	**JToggleButton**	**JToolTip**	**JTable**	**InternalFrame**
					
JRadioButtonMenuItem	**JSlider**	**JOptionPane**	**JScrollPane**	**JTextArea**	**JToolBar**