MAIN CLASS (saved as MainClassName.java)

```
    public class MainClassName {
        public static void main (String[] args) {
            //* main program goes here */          }   }
```

CLASSES

Declaring a Class : *modifiers* **class** class_name {
instance variables
constructors
methods }

Class Modifiers :

- **public** : gives public visibility
- **abstract** : a class that has abstract (empty) methods
- **final** : a class that can have no subclasses
- undefined = *friendly* : can be instantiated by other classes in the same package

INSTANCE VARIABLES

Declaring instance variables : *modifiers type* name**;**

Variable Modifiers

- Scope Modifiers
  - **public** : anything can access
  - **protected** : accessed only by same package
  - **private** : accessed only by same class
- Usage Modifiers
  - **static** : associated with class, not subclasses
  - **final** : has an unchangeable initial value

Variable Types

- Base Types (Primitive Types):
  - **boolean** : true/false
  - **char** : 16-bit Unicode --- Special chars : **\n** (new line), **\b** (backspace), **\f** (form feed), **\'** (single quote: '), **\t** (tab), **\r** (return), \\ (backslash:\), **\"** (double quote:")
  - Integers (signed, 2's complement)
  - **byte** : 8-bit, **short** : 16-bit, **int** : 32-bit, **long** : 64-bit
  - Floating Points (IEEE)
  - **float** : 32 bit, **double** : 64 bit
- Reference Types
  - defined class, **String**, **Array**, etc.
- Enum Types: *modifier* **enum** enum_name { value$_n$… };

CONSTRUCTORS

Constructors set the default values of the variables when a new instance of the class is created. The constructor has the same name as the class. More than one constructor can be present to handle various inputs.

Declaring Constructors : *modifiers* name (*type*$_n$ parameter$_n$…);

Constructor Modifiers: **public**, **protected**, **private**

METHODS

Declaring Methods : *modifiers return_type* method_name (*type*$_n$ parameter$_n$…) 1.
{ } 2.

Method Modifiers : **public**, **protected**, **private**, **abstract**, **static**, **final**

OPERATORS

Assignment =
Arithmetic: +, -, *, /, % (modulo/remainder)
Logicals <, <=, ==, !=, >=, > 1.
Boolean logicals! (not), && (and), || (or)

CONTROL FLOW

- IF : evaluates boolean
  if (boolean1) {…} **else if** (booleann} {…} **else** {…} 2.
- SWITCH : evaluates integer or enum

```
switch (i) {
    case a1:  …;  break;
    case an:  …;  break;          3.
    default: …;            }
```

LOOPS

WHILE :	**while** (boolean1) **{…}**
DO-WHILE:	**do {…} while** (boolean1)**;**
FOR:	**for (**initialization (*i=0*)**;** condition (*i<5*)**;** increment (*i++*) **) {…}**

CONTROL-FLOW STATEMENTS

return; exits a method. may return a value.

break; breaks out of innermost switch or loop, or out of labeled loop.

labeled loops:

```
loop1:
    for () {…}
```

continue; skip remaining steps in current iteration, and return to start of loop

ARRAYS

Declaring: <type>**[]** array_name **= new** <type>**[***10***]**

Declaring Higer dimension arrays:

- 2D: *type* **[][]** array_name**;**
- nD: *type* **[]$_n$** array_name**;**

Ragged arrays: **int[][]** a = **new** int[3][]**;** a[0] = **new** int[2]; etc

Looping through:

- **for** (*type* element_name **:** array_name) {…;}
- **for (int** i=**0;**i<**array_name.length;**i++) {…;}

Cloning Entire Arrays: *type*[] array_name2 = (*type*[]) array_name**.clone();**

Copying Parts of Arrays:

System.arraycopy(source_name, copy_pos, dest_name, paste_pos, count)**;**

ARRAYLISTS

Declaring: **java.util.ArrayList**<*type*> arraylist_name = **new ArrayList**<type>**();**

Methods: arraylist_name**.add** (,or: **.get**, **.remove**, **.set**, **.size**)

INPUT & OUTPUT

Output: **System.out.print (…); System.out.println (…);**

Input:
import java.io.*;
import java.util.Scanner;
Scanner scanner_name = new Scanner(System.in);

SCANNER METHODS

Tests (returns Booleans): **hasNext()** **hasNext*Type*()** **hasNextLine()**

Returns: **next()** **next*Type*()** **nextLine()**

String input_name = scanner_name**.nextLine();**
char c = input_name**.charAt(**0**);**
int i = **Integer.parseInt(**input_name); *(or **long**, **float**, **double**)*

FILE INPUT & OUTPUT

Input: **java.io.FileReader; File** source_name = **new File ("**FileName.txt**");**

Output: **java.io.PrintWriter; FileWriter** write_name = **new FileWriter ("**OutName.txt**", false);**

Printing:
java.io.PrintWriter; PrintWriter outFile=new **PrintWriter(fw, true);**
java.io.PrintStream; PrintStream ps= **new PrintStream(**"out.txt"**);**

File Methods: **.exists(); .canRead(); .canWrite(); .isFile(); .isDirectory(); .close(); .flush();**

System. methods: **System.err; System.exit(1);**

NESTED CLASSES & PACKAGES

Nested Classes: closely related classes, saved in the same .java file. The nested class should be declared **static**.

Packages: a set of classes defined in a common subdirectory.

- **package** *package_name*; **import** *packageName.classNames*;

JAVA DOCS

```
/** What to do.
    @param paraName description of paraName
    @return description of return            */
```

OBJECT ORIENTED DESIGN GOALS

Robust – capable of handing unexpected inputs

Adaptable – can evolve over time in response to changing conditions and can be run on different hardware (portable)

Reusable – same code should be useable as a component on different systems

OBJECT ORIENTED DESIGN PRINCIPLES

Abstraction – distill a complicated system to its most fundamental parts. Name those parts and explain their functionality. Classes use Abstract Data Types (ADTs) as Interfaces.

Encapsulation – different components of a software system should not reveal their internal implementation details. Advantage: Programmer has freedom to implement the details of a system, but maintains the abstract interface seen by outsiders.

Modularity – different components of a software system are divided into functional units.

INHERITANCE

- Hierarchical structure for re-using code.
- **Child class inherits all protected and public methods and data members from Parent.** Sibling classes are not equivalent and cannot access each others' methods and data members. A Child can only have (extend) one parent.

Implementing Inheritance:

```
public class Parent_name { … }
public class Child_name extends Parent_name { … }
```

POLYMORPHISM

Overriding Inherited Methods: a child class explicitly defines a method with the same name, inputs & return type as the parent class

OVERLOADING

Overloading Methods: Simultaneous definitions of a method that have the same name, but that handle different kinds of inputs.

ABSTRACT METHOD A method declaration with no body.

```
public abstract ret_type method_name1();
```

ABSTRACT CLASS A class that contains a mix of concrete and abstract methods. Abstract classes cannot be instantiated, and must be **extended**.

```
abstract class abstract_name { … }
```

INTERFACES

A collection of method declarations with no instance data and no bodies. Interface classes cannot be instantiated and must be **implemented**.

Declaring an Interface:

```
public interface face_name {
   public ret_type method_name1();
   public ret_type method_namen();   }
```

Implementing an Interface:

```
public class class_name implements face_name {
   public ret_type method_name1( …);
   public ret_type method_namen( …);              }
```

A class can implement multiple interfaces:

```
public class class_name implements face_name1, face_name2 { …}
```

COMBINING INHERITANCE AND INTERFACES

```
public class Child_name extends Parent_name implements Face_name{ }
```

INHERITANCE KEYWORDS

- **this()** : refers to the current instance of a class. Also references an object field if the name clashes with a field in a block.
- **super()** : explicitly calls the parent's constructor

PARENTS WITH MULTIPLE CHILDREN

Suppose a parent class Student has been extended to two child classes: UndergradStudent and GradStudent. The Parent-class reference can refer to child-class types:

```
Student x = new UndergraduateStudent();
Student y = new GradStudent();
if ( x instanceof GradStudent) { … }  ← this tests for the correct child type
```

EXCEPTIONS

Unexpected events that occur when a program is run.

Throwing Exceptions:

```
throw new exception_type(param0, param1, … paramn-1);
```

THROWS clause:

```
   public void class_name() throws exception_name1, exception_name2 {…}
```

Exception and Error are subclasses of Throwable (which denotes any object that can be thrown or caught). Specialized exceptions can be defined by subclassing from either Exception or RuntimeException.

Catching Exceptions: When an exception is thrown, it must be caught (and the problem dealt with) or the program terminates.

try-catch block:

```
try { … }
catch(exception_type1 variable1){ … }
catch(exception_typen variablen){ … }
catch(exception_type2 e){
   system.out.println(e.getMessage());
   e.printStackTrace();
}
finally{ … }
```

CASTING (Type conversion)

Widening: Citrus c = new Orange(); //store the child as parent type

Narrowing: Orange o = (Orange) c;   //force a parent into a child type

GENERICS

Types that are not defined at compilation, but are defined at runtime. i.e:

Declaring Generics:

```
public class Pair<K,V>{            // K & V represent generic types
   K key;  V value;
   public void setKey(K k, V v){  key = k;  value = v;    }  }
```

Specifying Generics:

```
public static void main (String[] args) {
   Pair<String, Integer> pair1 = new Pair<String, Integer> ();
   Pair<Student, Double> pair2 = new Pair<Student, Double > ();
```

ENUMERATORS (an example)

```
public enum Month{JAN, FEB, MAR}
System.out.println(Month.JAN) // JAN
Month m = Month.valueOf("MAR")  // m = Month.MAR
System.out.println(Month.JAN = m); // false
System.out.print(m); // MAR
```

ASSERTIONS

```
Javac –source 1.4 Filename.java  // to compile a class with assertions
java –ea filename                // to turn on assertion at command prompt
assert x >=0 : "I'm smart";      // if x>0, print "I'm smart"
```

USEFUL CLASSES

Math Class

java.lang package

floor(a), max(a,b), min(a,b), pow(a,b), random(), toDegrees, toRadians, abs(a)

GregorianCalendar CLASS

java.util.Date

GregorianCalaendar IndependenceDay = new GregorianCalendar(1776, 6, 4);

IndependenceDay.getTime();

YEAR, MONTH, DATE, DAY_OF_YEAR, DAY_OF_MONTH, WEEK_OF_YEAR, AM_PM, HOUR, HOUR_OF_DAY, MINUTE

DecimalFormat CLASS

java.text package

DecimalFormat df = new DecimalFormat("0.000");

Date and SimpleDateFormat CLASS

java.text package

Date today = new Date();

SimpleDateFormat sdf = new SimpleDateFormat("EEEE"); // Shows Saturday

JOptionPane.showMessageDialog(null, "Today is" + sdf.format(today));

JOptionPane.showInputDialog(null, "Enter text:");

yyyy: 2002; MM: 10; MMM: Oct; MMMM: October; dd: day in mth; DDD: day in yr; hh: hr; HH: 24 hr; mm: Minute; ss: Seconds; mmm: Millisec; E: Sat; EEEE: Saturday

String Class

substring(), length(), indexOf(), equals(), compareTo(), charAt(), toUpperCase(), Integer.parseInt();

StringBuffer Class

```
StringBuffer word = new StringBuffer("Java")
word.setCharAt(0,'D');
word.append(word);
word.insert(index, "String");
```

StringTokenizer Class

```
StringTokenizer parser = new StringTokenizer(inputString);
while(parser.hasMoreTokens()){… … …}
type = parser.nextToken();
```