Introduction classes and Objects

What are classes?

Java is an object oriented programming language so, any java programe consist of one and more class definitions.

In the real world, you often have many objects of the same kind. for example , your TV is just any of many TV's in the world. using object oriented programming we say that your TV object is instance of the class of objects known as Televisions.

What are classes?

Television have some state

-screen on

-screen off

-volume control

And behavior

-change channels

-colors

In common

What are classes?

However, each TV's state is independent of and can be different form other TV's. Every system must be visualized as a class of objects.

class

A class is usually described as the blueprint or template from which, the object is actually made. Its attributes and methods represent a class

Declaring a class

A class is described by use of the **class** keyword. The attributes and methods (behavior) of a class are defined inside a class body. In java, the braces {}mark the beginning and the end of a class or method

Declaring a class

Syntax:

[access- specifier] [modifier] class <class name>

{

data type instance-variable1;

data type instance-variable1;

……..

//methods

}

[]-are optional

<>-are mandatory (must)

Declaring a class

Example:

```
Class Tv
{
//data members
//methods
}
```

Object

An object is an instance of class. A software object maintains its states in variables and implements its behavior with methods declaring on object is similar to declaring a variable.

Syntax:

<class-name><object-name>;

E.g.,

Tv sony;//declare

Sony=new Tv();//create

Or Tv sony=new Tv();//declare and create

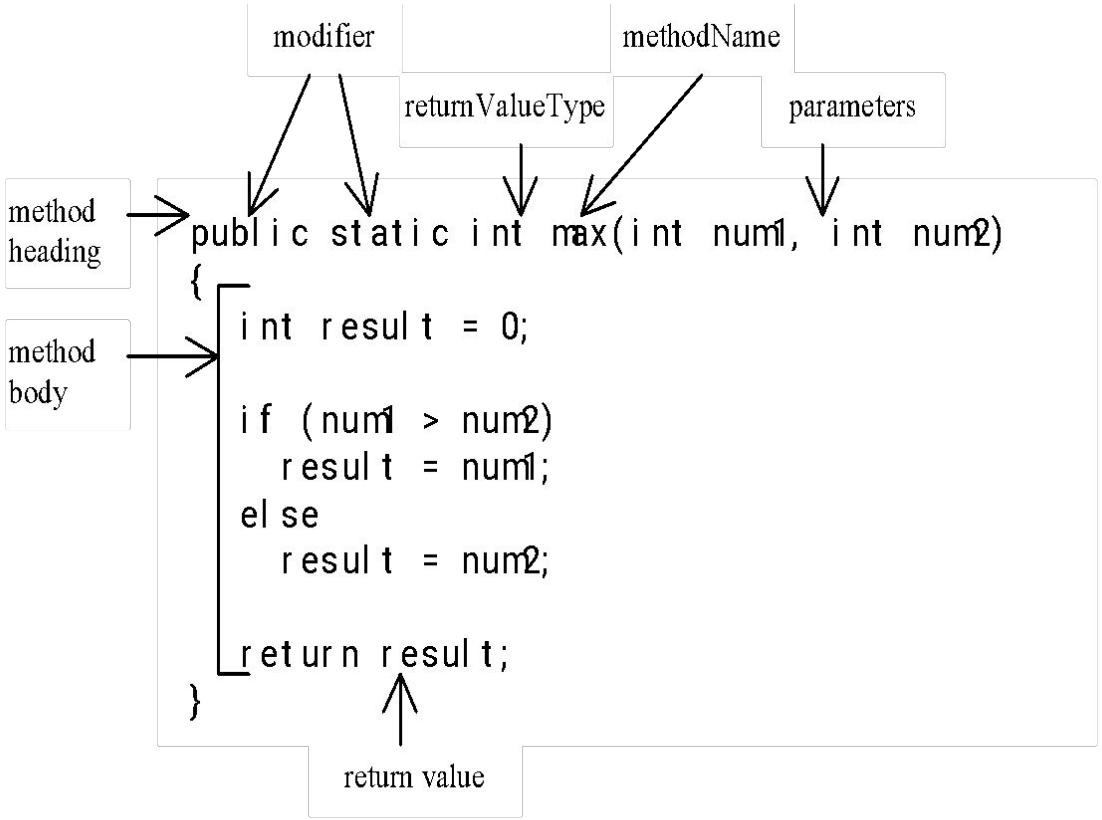
Methods

- Introducing Methods
- Declaring Methods
- Calling Methods
- Passing Parameters
- Pass by Value

Introducing Methods

A method is a collection of statements that are grouped together to perform an operation.

Method Structure



Declaring Methods

```
public static int max(int num1,
  int num2)
{
   if (num1 > num2)
      return num1;
   else
      return num2;
}
```

Calling Methods

- Call by value
- Class by reference

Call value

```
Class my function
 {
Void change(int i)
{
i=i*20;
System.out.println(“inside method i=“+i);
}}
Class callbyvlaue{
Public static void main(String args[]){
Myfunction myf=new myfunction();
int x=20;
System.out.println(“Before call x=“+x);
myf.change(x);//calling here
System.out.println(“After call x=“+x);}}
```

Characteristics of call by value

1. Original value can not be changed.
2. Called method gets only a copy of the variable.
3. In java, all arguments of the primitive data type are passed by value.
4. The copy of the argument value is maintained at a separate memory location.

Call by reference

```
Class sampleTest{
Int i;
sampleTest(int j){i=j;}
Void change(sampleTest s){
s.i=s.i*20;
System.out.println(“Inside method i=“+s.i);}}
Class callbyref{
Public static void main(String args[]){
sampleTest s1=new sampleTset(20);
System.out.println(“Before call i=“+s1.i”);
S1.change(s1);
Syste.out.println(“After call i=“+s1.i);}}
```

constructors

A constructor is a member function of a class that is used to create objects of that class. It has the same name as the class itself, it has no return type, and is invoked using new operator. A constructor is a special kind of function which is used only to create the class's object.

constructors

Syntax:

(modifier)(classname)(parameter list)

{

//body

}

Types of constructors

1. Default constructor
2. Argument constructor
3. Copy constructor

1. Default constructor

The simple kind of constructor that a class can have is the one that has no parameter. When you do not explicitly defined a constructor for class, then java, create a default constructor for the class. The default constructor automatically initialize all instance variables.

General form:

classname class-variable(or) objectname=new classname();

e.g., Triangle t1=new Triangle();

Default constructor programe

```
Class add{
int a,b;
add(){
System.out.println(“Default Constructor”);
a=10,b=20;}
void sum(){
System.out.println(“a+b=”+(a+b));}}
class addition{
public static void main(String args[]){
Add test=new add();
test.sum();}}
```

2. Argument constructor

In this type of constructor parameters are passed as a argument.

This is also called parameterized constructors.

Argument constructor programe

```
Class Triangle{
Double base,  height;
Triangle (double b, double h){
Base=b, height=h;}
Double area(){
Return(0.5*base*height);}}
Class triangle1{
public static void main(String args[]){
Double area1;
Triangle t1=new Triangle(10,20);
Triangle t2=new Triangle(20,30);
area1=t1.area();
System.out.println(“Area of t1=“+area1);
area1=t2.area();
System.out.println(“Area of t2=“+area1);
```

3. Copy Constructor

A copy constructor is a constructor that replicates duplicates an existing object. So it is called the copy constructor. It takes a parameters is a reference to an object of the same class to which the constructor depend.

Syntax:

Classname(classname copyconstructorname)

e.g., A a=new A();

A b=new A(a);

Copy Constructor programe

```
Class copy{
int a, b;
Public copy(int a1,int b1){
a=a1,b=b1;}
Public copy(copy c){//copy constructor
a=c.a,b=c.b;}
Void view(){
System.out.println(“a=“+a”b=“+b);}}
Class copytest{
public static void main(String args[]){
copy test=new copy(10,15);
System.out.println(“First Constructor”);
test.view();
Copy test1=new copy(test)//copy constructor called
System.out.println(“Second Constructor”);
Test1.view();}}
```

This Key Word

1. This is used for identify the object to which class it belongs.

2. It is also used for calling current objects.

3. This is also used for making difference between instance variable and local variable .

Programe for this key word

```
Class thistest{
String s="Instance variable";
Void view(){
String s="Local variable";
System.out.println("s="+s);
System.out.println("This s="+this.s);}
Public static void main(String args[]){
Thistest t=new thistest();
t.view();}}
```

Garbage Collection

The JVM's heap stores all objects created by an executing Java program. Objects are created by Java's "new" operator, and memory for new objects is allocated on the heap at run time.

Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs and headaches.

Why Need Garbage Collection

Garbage collection relieves programmers from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the JVM has several advantages.

First, it can make programmers more productive.When programming in non-garbage-collected languages the programmer can spend many late hours (or days or weeks) chasing down an elusive memory problem. When programming in Java the programmer can use that time more advantageously by getting ahead of schedule or simply going home to have a life.

A second advantage of garbage collection is that it helps ensure program integrity. Garbage collection is an important part of Java's security strategy. Java programmers are unable to accidentally (or purposely) crash the JVM by incorrectly freeing memory.

The Finalize() method

finalize Method

-Call the finalize method before the garbage collector reclaim the memory

Provide the method to release the resources

- **Programmer can remove the resources (ex:open files) directly using finalize method which garbage collector cannot reclaim**

Programe on Garbage Collection

protected void finalize() throws Throwable {
// … }

```
public class GcTest{
static {
  System.out.println("Initialize");
}

protected void finalize() throws Throwable {
  System.out.println("Clean Up");
}

public static void main(String args[]) {
  System.out.println("In main .... ");
  GcTest x = new GcTest();
  x = null;
  System.gc();  // We need to call this point to see the "Clean Up"
}
}
```

Overloading Constructors

- If you create a class from which you instantiate objects, Java automatically provides a constructor
- But, if you create your own constructor, the automatically created constructor no longer exists
- As with other methods, you can overload constructors
  - Overloading constructors provides a way to create objects with or without initial arguments, as needed

Overloading Constructors

- Like overloaded methods, there exists overloaded constructors too.

// eg. of an overloaded constructor

```
Box ob= new Box();
Box obj1= new Box(len);  //square
Box obj2= new Box (w, h, d);//rectangle
```

OBJECTS AS PARAMETERS

```
// RETURNS TRUE, o IS EQUAL  TO THE
   INVOKING OBJECT

   boolean equals(Test o)
   {
    if(o.a==a && o.b==b)
            return true;
    else return false;      }
```

RECURSION

- Recursion is the process of defining something in terms of others.
- An *attribute,* that allows a method to call itself.
- A method that calls itself is said to be recursive

ACCESS SPECIFIERS

Access Levels

Specifier	Class	Package	Subclass	World
private	*Y*	*N*	*N*	*N*
no specifier	*Y*	*Y*	*N*	*N*
Protected	*Y*	*Y*	*Y*	*N*
Public	*Y*	*Y*	*Y*	*Y*

static KEYWORD

- ***static*** is used to define a class member described independently of any objects of that class.
- **Limitations:**

  -- can call only static methods

  -- can access only static data

  -- they can't refer **this / super** in any way.

final KEYWORD

- A variable declared as *final* essentially behave as a constant.
- These variable do not occupy memory on per instance basis.

eg.

```
final int FILE_OPEN= 1;
```

NESTED CLASSES

The Java programming language allows you define a class within another class. Such a class is called a *nested class* and is illustrated here:

EG.

```
class EnclosingClass {
    ...
    class ANestedClass { ... }
}
```

COMMAND-LINE ARGUMENTS

When we pass information, i.e. arguments into a program when we execute it, it is known as *command-line arguments.*

eg.
```
class commline{
    public static void main(String[] args)
    {    for(int i=0; i<args.length; i++)
         System.out.println("args["+i"]:+args[i]);
    }
```

COMMAND-LINE ARGUMENTS (contd.)

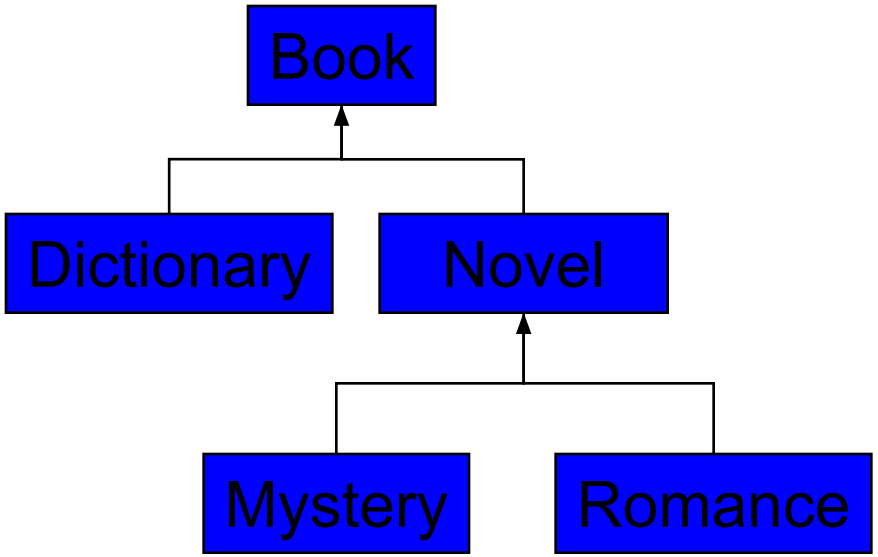
- Each of the elements in the array named **args** (including the element at position zero) is a reference to one of the *command-line arguments* each of which is a **String** object.
- The number of elements in a Java array can be obtained from the **length** property of the array.
- Therefore, it is not necessary for the operating system to also pass a parameter specifying the number of arguments.
- *Command-line-arguments* are used in many programming and computing environments to provide information to the program at startup that it will need to fulfill its mission during that particular invocation

INHERITANCE

INHERITANCE

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

  The subclass is a more specific version of the Original
- (Remember has-a is aggregation



INHERITANCE

- The child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

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

```
class Dictionary extends Book {

    // class contents
}
```

```
Dictionary webster = new Dictionary();
webster.message();
webster.defMessage();
```

Number of pages: 1500

Number of definitions: 52500

Definitions per page: 35

```
public class Book {

    protected int pages = 1500;

    public String message() {
        System.out.println("Number of pages: " + pages);
    }
}

public class Dictionary extends Book {

    private int definitions = 52500;

    public void defMessage() {
        System.out.println("Number of definitions" +
                           definitions);
        System.out.println("Definitions per page: " +
                           (definitions/pages));
    }
}
```

Some Inheritance Details

- An instance of a child class does not rely on an instance of a parent class
  - Hence we could create a Dictionary object without having to create a Book object first
- Inheritance is a one-way street
  - The Book class cannot use variables or methods declared explicitly in the Dictionary class

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

```
public class Book {
 protected int pages;

    Book(int numPages) {
        pages = numPages;
    }
}
public class Dictionary {

    private int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }
}
```

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead (will discuss later)

Overriding Methods

- When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version overrides the parent's version in favor of its own.
  - Signature: method's name along with number, type, and order of its parameters
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

```
public class Book {
   protected int pages;
   Book(int numPages) {
      pages = numPages;    }
   public void message() {
      System.out.println("Number of pages: " + pages);  }}
public class Dictionary {
   protected int definitions;
   Dictionary(int numPages, int numDefinitions) {
      super(numPages);
      definitions = numDefinitions;    }
   public void message() {
      System.out.println("Number of definitions" +
                         definitions);
      System.out.println("Definitions per page: " +
                         (definitions/pages));
      super.message();    }
```

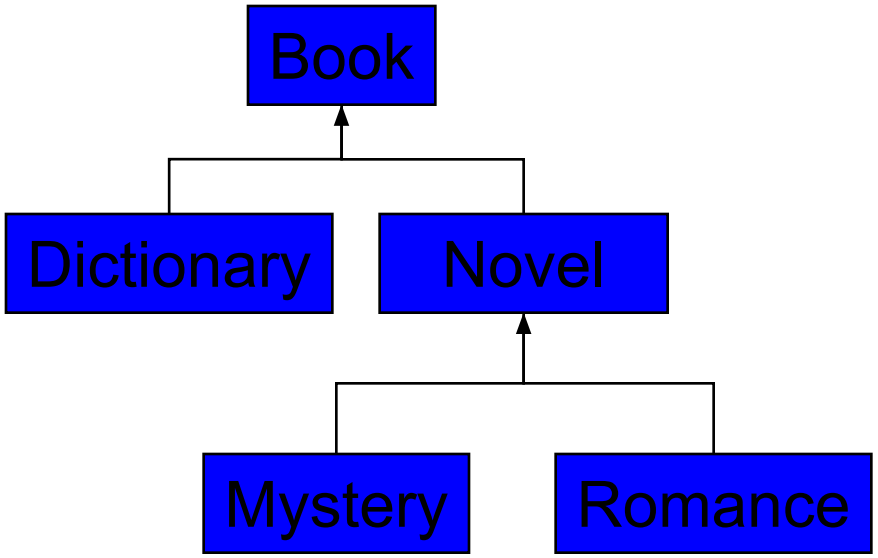
METHOD OVERRIDING

A *method* is overridden if there is a *method* in the *subclass* that has the same name and the **same** set of parameters. The *superclass method* is then NOT inherited.

```
// method overriding
class A{
 void showtest(){ //..} }
class B extends A {
 void showtest(){  //..}    }
```

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
  - *However they are not related by inheritance because one is not used to derive another.*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information
  - All objects are guaranteed to have a `toString` method via inheritance, thus the `println` method can call `toString` for any object that is passed to it

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

ABSTRACT CLASSES

These are superclasses that declares the structure of a given abstraction without providing the complete detailing.

:- ***subclass responsibility***

- no instance creation
- can’t declare abstract constructors, abstract static methods

Rules for Abstract classes

1. An abstract method is a method that is declared with only signature it has no implementation.
2. An abstract class is a class that has at least one abstract method.

   abstract class classname{

Abstract methodname(parameterlists);

Ordinary methods;}

3. You can not declare any abstract constructor.
4. Abstract class has not include any abstract static method.
5. Concrete methods are also be inside the any abstract class.

Programe for Abstract class

```
abstract class operation
{   int a,b;
    operation(int a1,int b1)
    {a=a1;
     b=b1;
    System.out.println(a+""+b
    );     }
    abstract int result();
}
```

```
class add extends operation
{   add(int a1,int b1)
    {super(a1,b1);}
    int result()
    {
    System.out.println("Inside
    Add-Addition");
    return(a+b);}}
```

Programe for Abstract class

```
class subtract extends operation{
   subtract(int a1,int b1)
   {   super(a1,b1); }
   int result()
   {   System.out.println("Iside
   subtract");
return(a-b);}}
```

Programe for Abstract class

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

    add a1=new add(3,2);
    subtract s1=new subtract(3,2);
    operation ref;
    ref=a1;
    System.out.println("Result is"+ref.result());
    ref=s1;
    System.out.println("result is"+ref.result());}}
```

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

final :

- **// to prevent overiding**

  a method declared as final can't be overrided
- **// to prevent inheritance**

  a class declared as final, can't be inherited further.