Bharathidasan University

Centre for Differently Abled Persons

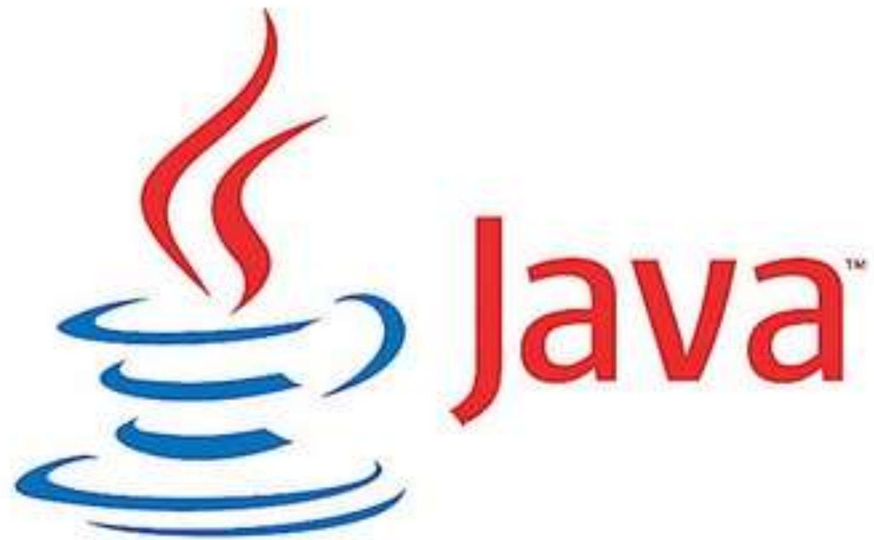
Tiruchirappalli - 620024.

- Programme Name: Bachelor of Computer Applications
- Course Code : 23UCACC04
- Course Title : Programming in Java
- Semester : IV
- Unit : Unit IV
- Compiled by : Dr. M. Prabavathy

Associate Professor

Ms. M. Hemalatha

Guest Faculty


Java™

EXCEPTION
HANDLING

Exception Handling

- The Exception Handling in Java handle the **runtime errors** so that normal flow of the application can be maintained.
- An exception is an **unwanted or unexpected event**.

Try and Catch Block

The **try** statement test **block of code for errors** while it is being executed.

The **catch** statement executes block of code**, if an error occurs in the try block.**

The try and catch keywords come in pairs.

Syntax

```
try
{
    // Block of code to try
}
catch(Exception e)
{
    // Block of code to handle errors
}
```

Example

```
public class Main
{
   public static void main(String[] args)
   {
         int[] myNumbers = {1, 2, 3};
         System.out.println(myNumbers[10]);
  }
}
```

It throws ArrayIndexOutOf Bound Exception.

USE OF THROW KEYWORD

The throw statement allows you to create a **custom error.**

The throw statement is used together with an exception type.

Throw is used within the method.

Cannot throw multiple exceptions.

Arithmetic Exception,
FileNotFoundException,
ArrayIndexOutOfBoundsException,
SecurityException, etc…,

Example

```
public class TestThrow1
{
static void validate(int age)
{
    if(age<18)
throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
   }
public static void main(String args[])
{
validate(13);

  }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:not valid

USE OF THROWS KEYWORDS

Java **throws** keyword is used to **declare an exception.**

Throws is followed by class.

Can declare multiple exceptions.

Example

```
void m()throws ArithmeticException
{
      //method code
}
```

FINALLY KEYWORD

Java finally block is always **executed whether exception is handled or not.**

Java finally block follows try or catch block.

Example

```
import java.io.*;
classTestFinallyBlock
{
public static void main(String args[])
{
Try
{
int data=25/5;
```

```
System.out.println(data);
  }
catch(NullPointerException e)
{
System.out.println(e);
}

finally{
System.out.println("finally block is always executed");
}

System.out.println("rest of the code...");
  }
}
```

OUTPUT:

```
5
finally block is always executed
rest of the code.
```

USER DEFINED EXCEPTION

- **Creating own Exception** that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.

Example

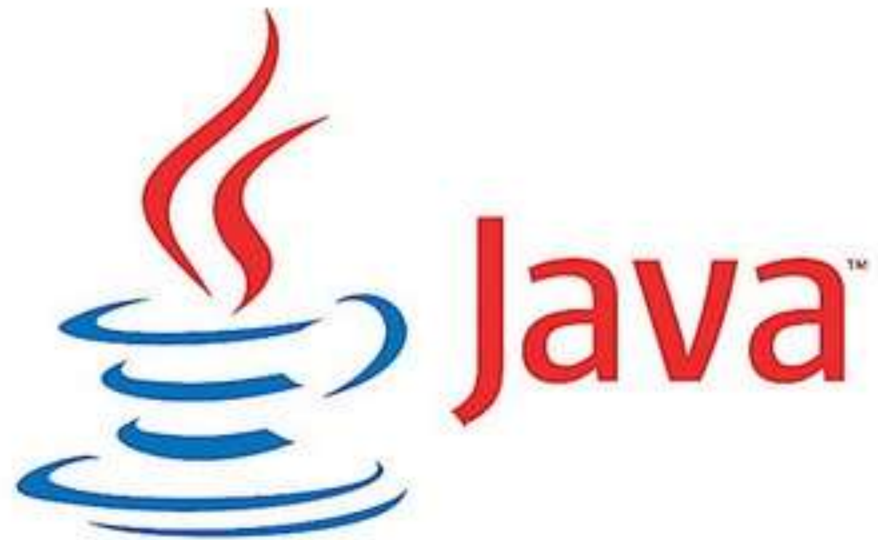
```
classJavaException
{
public static void main(String args[])
{
```

```
try
{
   throw new MyException(2);
  }
catch(MyException e)
{
   System.out.println(e) ;
 }
}
}
classMyException extends Exception
{
   int a;
```

```
MyException(int b)
{
    a=b;
  }
public String toString()
{
return ("Exception Number =  "+a) ;
 }
}
```

Output:

Exception Number = 2


Java™

INPUT – OUTPUT
(FILES)

Files

The File class is an **abstract representation of file** and **directory pathname**.

The File class have several methods for working with directories and files such as

- **creating** new directories or files
- **deleting** files or directories
- **Renaming** directories or files
- **listing** the contents of a directory etc.

FILE PERMISSION

Java FilePermission class contains the permission related to a directory or file.

All the permissions are related with path.

The path can be of two types:

1) **D:\\IO\\-:** permission with all sub directories and files.

2) **D:\\IO*:** permission is associated with all directory and files within this directory excluding sub directories.

A file can be in any combination of following permissible permissions:

1. Executable:

Tests whether the application can **execute the file** denoted by this abstract path name.

2. Readable:

Tests whether the application can **read the file** denoted by this abstract path name.

3. Writable:

Tests whether the application can **modify the file** denoted by this abstract path name.

CREATE A FILE

To create a file in Java, use method

createNewFile()

This method returns a boolean value:

True if the file was successfully created

False if the file already exists.

Example:

```
public class CreateFile
{
    public static void main(String[] args)
    {
        Try
        {
            File myObj = new File("filename.txt");
```

```
            if (myObj.createNewFile())
            {
                        System.out.println("File created: " +
                                                        myObj.getName());
            }
             else
            {
                         System.out.println("File already exists.");
            }
        }
         catch (IOException e)
          {
                System.out.println("An error occurred.");
                 e.printStackTrace();
             }
        }
}
```

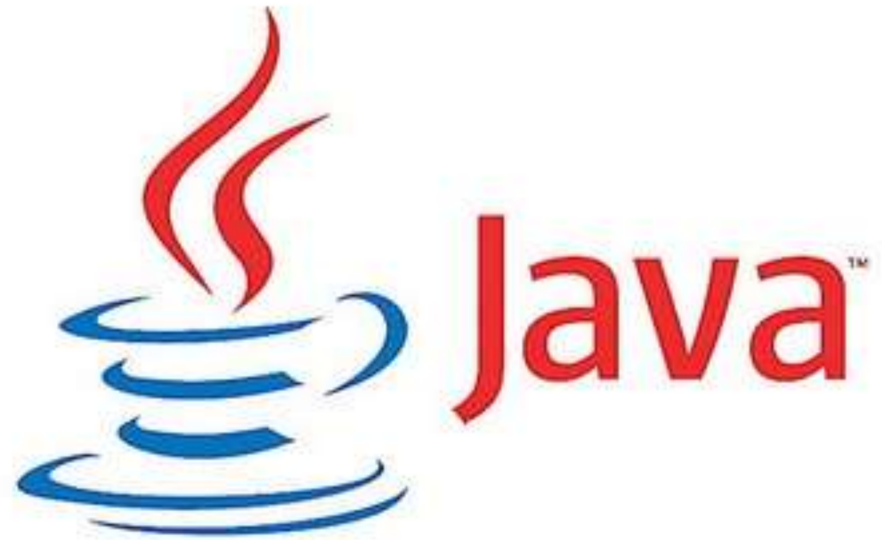
WRITE TO A FILE

- Use the FileWriter class together with its **write()** method to write some text to the file.
- Writing to the file should always have **close()** method with it.

Example:

```
public class WriteToFile {
public static void main(String[] args) {
try
{
 FileWriter myWriter = newFileWriter("filename.txt");
```

```
myWriter.write("Files in Java might be tricky, but it is fun
   enough!");
   myWriter.close();
   System.out.println("Successfully wrote to the file.");
    }
catch (IOException e)
{
         System.out.println("An error occurred.");
         e.printStackTrace();
  }
 }
}
```


Java™

INPUT – OUTPUT
(STREAM)

Input-Output

- Java I/O (Input and Output) is used **to process the input and produce the output.**
- Java uses the concept of a stream to make I/O operation fast.
- The **java.io** package contains all the classes required for input and output operations

STREAM

A stream is a **sequence of data**.

In Java, a stream is composed of bytes.

It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically.

1) **System.out:** standard output stream
2) **System.in:** standard input stream
3) **System.err:** standard error stream

OUTPUTSTREAM CLASS

OutputStream class is an **abstract class**.

It is the **superclass of all classes** representing an **output stream** of bytes.

An output stream accepts output bytes and sends them to some sink.

Methods of OutputStream

1. public void write(int)throws IOException

write a byte to the current output stream.

2. public void write(byte[])throws IOException

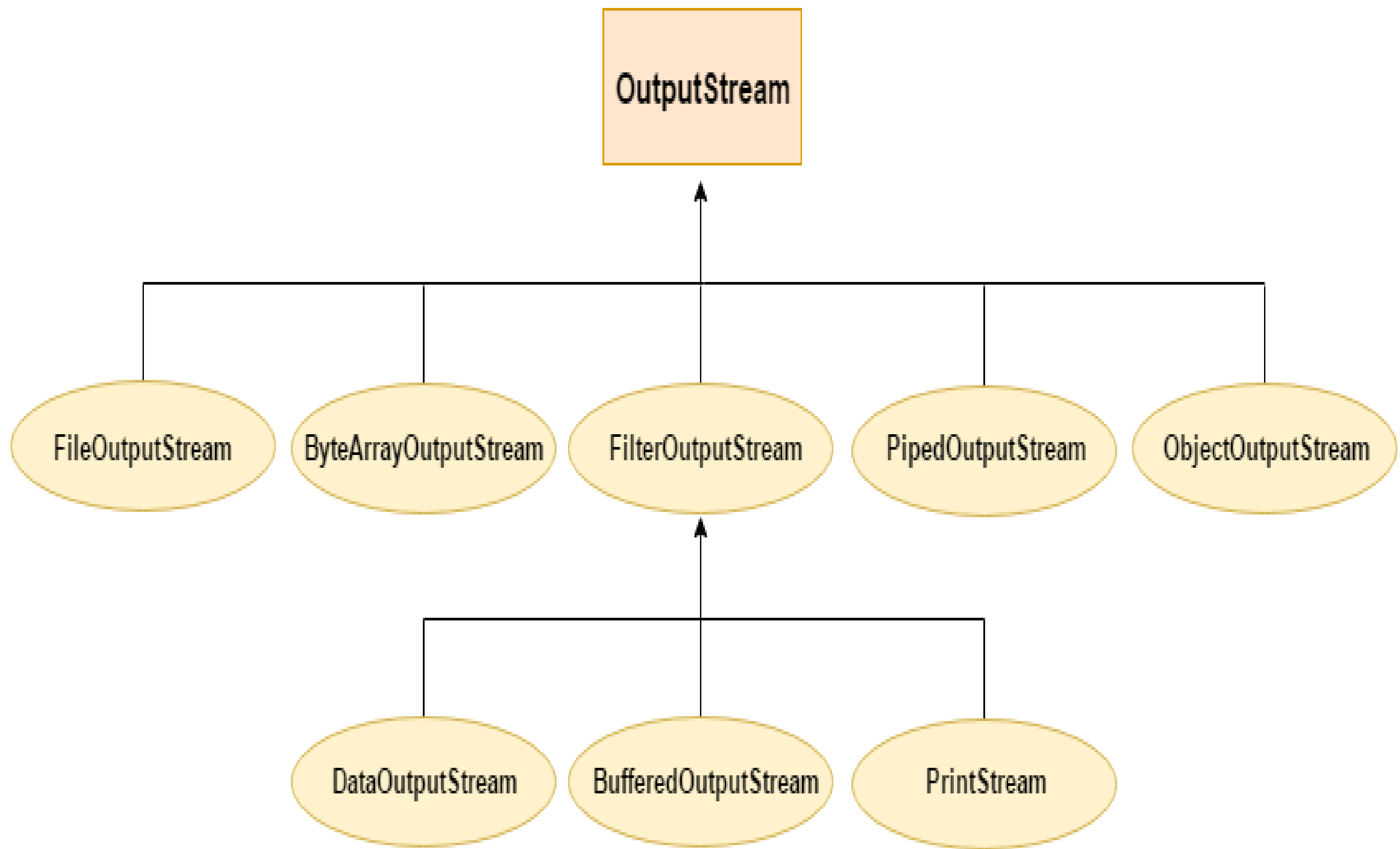
write an array of byte to the current output stream.

3.public void flush()throws IOException

flushes the current output stream.

4. publicvoid close()throws IOException

close the current output stream.


OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
PipedOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

INPUTSTREAM CLASS

InputStream class is an **abstract class**.

It is the **superclass of all classes** representing an **input stream of bytes**.

Methods of InputStream

1. public abstract int read()throws IOException

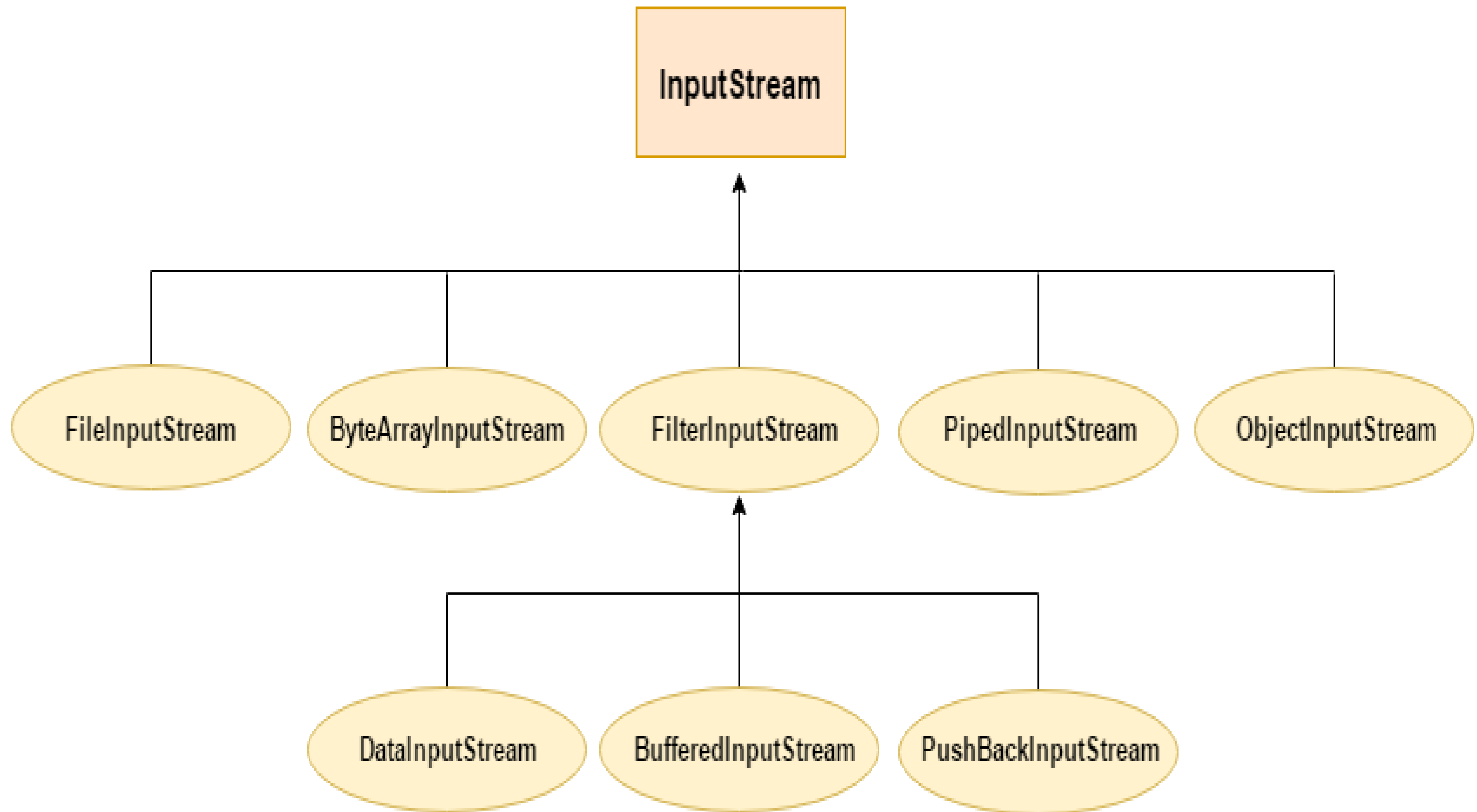
- reads the next byte of data from the input stream.
- It returns -1 at the end of the file.

2) public int available()throws IOException

- returns an estimate of the number of bytes that can be read from the current input stream.

3) public void close()throws IOException

- used to close the current input stream.


InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
PipedInputStream
ObjectInputStream
DataInputStream
BufferedInputStream
PushBackInputStream

MULTITHREADING

Multithreading

Multithreading in Java is a **process of executing multiple threads simultaneously.**

A thread is the **smallest unit of processing.**

Each of this process can be assigned either as a single thread or multiple threads.

Java Multithreading is mostly used in **games, animation, etc.**

Example: Single Thread

```
package demo;
 public class thread
{
public static void main(String[] args)
   {
          System.out.println("Single Thread");
    }
}
```

Life cycle of thread

There are various stages of life cycle of thread

1. New

- The thread is created using class "Thread class".
- It remains in this state till the program starts the thread.
- It is also known as **born thread**.

2. Runnable

- The **instance of the thread** is invoked with a start method.
- The thread control is given to scheduler to finish the execution.
- It depends on the scheduler, whether to run the thread.

3. Running

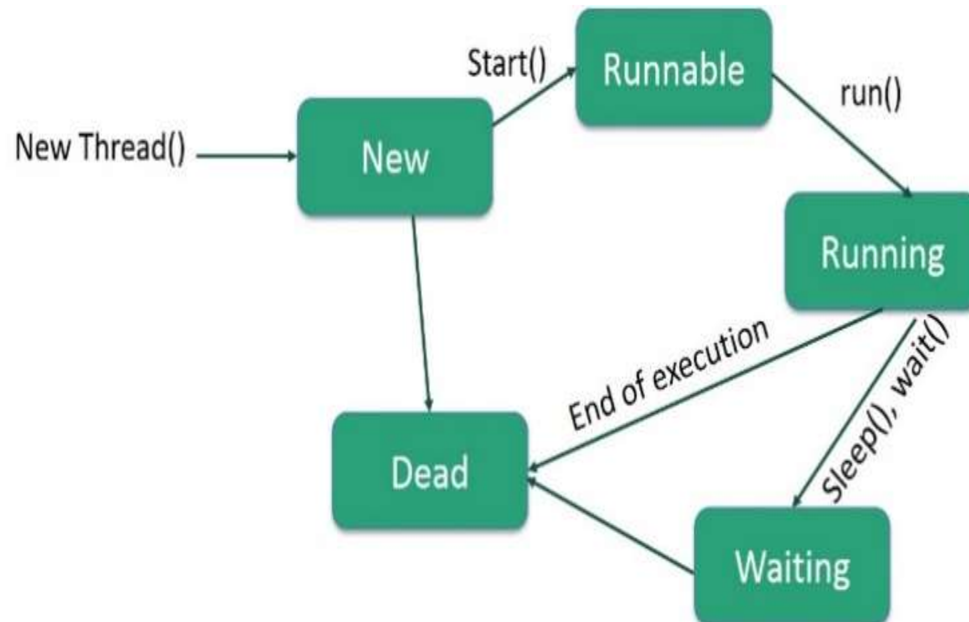
- When the **thread starts executing**, then the state is changed to "running" state.
- The scheduler selects one thread from the thread pool, and it starts executing in the application.

4. Waiting

- **one thread has to wait, till the other thread gets executed**.
- Therefore, this state is referred as waiting state.

5. Dead

- This is the state when the **thread is terminated.**
- The thread is in running state and as soon as it completed processing it is in "dead state”.



CREATING A THREAD

1. By extending Thread class
2. By implementing Runnable interface

Method used in Thread

getName(): Obtaining a thread's name

getPriority(): Obtain a thread's priority

isAlive(): Determine if a thread is still running

join(): Wait for a thread to terminate

run(): Entry point for the thread

sleep(): suspend a thread for a period of time

start(): start a thread by calling its run() method

Example:

```
importjava.awt.*;
importjava.applet.*;
public class car1 extends Applet implements Runnable
{
Thread m=null;
int p;
public void start()
{
m=new Thread(this);
m.start();
}
```

```
public void run()
{
    while(true)
    {
    for(p=30;p<getSize().width;p+=5)
    {
            repaint();
            try
            {
                    m.sleep(100);
            }
            catch(InterruptedException e)
            { }
    }}
}
public void stop()
{
m.stop();
m=null;
}
```

```
    public void paint(Graphics g)
    {
            g.setColor(Color.blue);
            g.drawString("WELCOME",85,10);
            g.setColor(Color.red);
            g.drawLine(p,10,p+40,10);
            g.drawLine(p-5,20,p+45,20);
            g.drawLine(p,10,p-5,20);
            g.drawLine(p+40,10,p+45,20);
            g.setColor(Color.black);
            g.fillOval(p,20,10,15);
            g.fillOval(p+30,20,10,15);
    }
}

//<applet code=car1.class width=400 height=500></applet>
```