Java^{TM} Education & Technology Services

Java Programming

Course Outline

- **Lesson 1:** Introduction to Java
- **Lesson 2:** Basic Java Concept
- **Lesson 3:** Applets
- **Lesson 4:** Data Types & Operators
- **Lesson 5:** using Arrays & Strings
- **Lesson 6:** Controlling Program Flow

Presentation Outline

- Lesson7: Java Exception
- Lesson 8: Interfaces
- Lesson 9: Multi-Threading
- Lesson 10: Inner class
- Lesson 11: Event Handling

Lesson 1

Introduction To Java

Brief History of Java

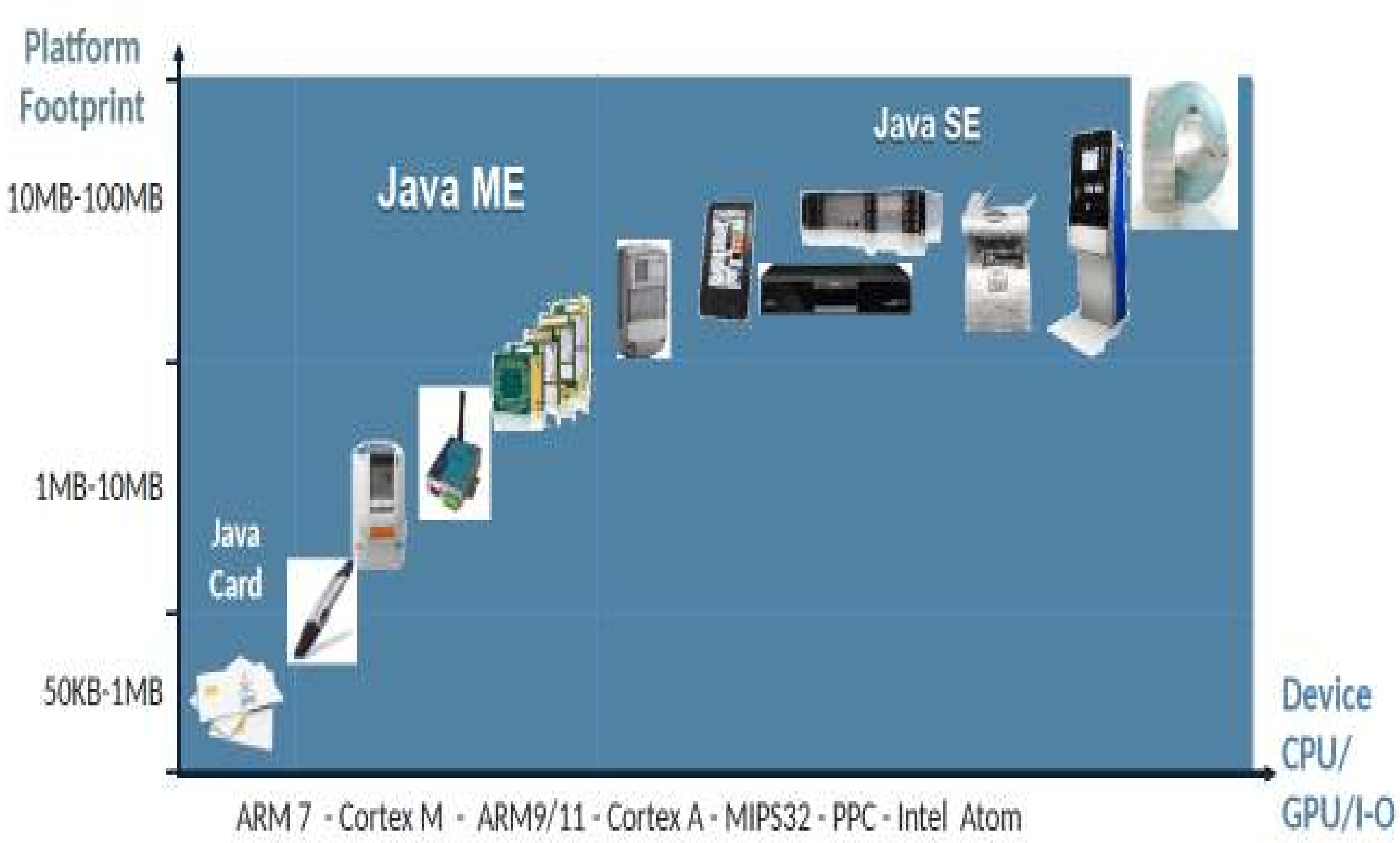
- Java was created by Sun Microsystems in **may 1995.**
- The Idea was to create a language for controlling any hardware, but it was too advanced.
- A team - **that was called the Green Team** - was assembled and lead by **James Gosling**.
- Platform and OS **Independent** Language.
- **Free** License; cost of development is brought to a minimum.

Brief History of Java

- From mobile phones to handheld devices, games and navigation systems to e-business solutions, **Java is everywhere!**

- Java can be used to create:
  - Desktop Applications,
  - Web Applications,
  - Enterprise Applications,
  - Mobile Applications,
  - Smart Card Applications.
  - Embedded Applications (Raspberry PI)
  - Java SE Embedded

Java is EveryWhere



Java Principles

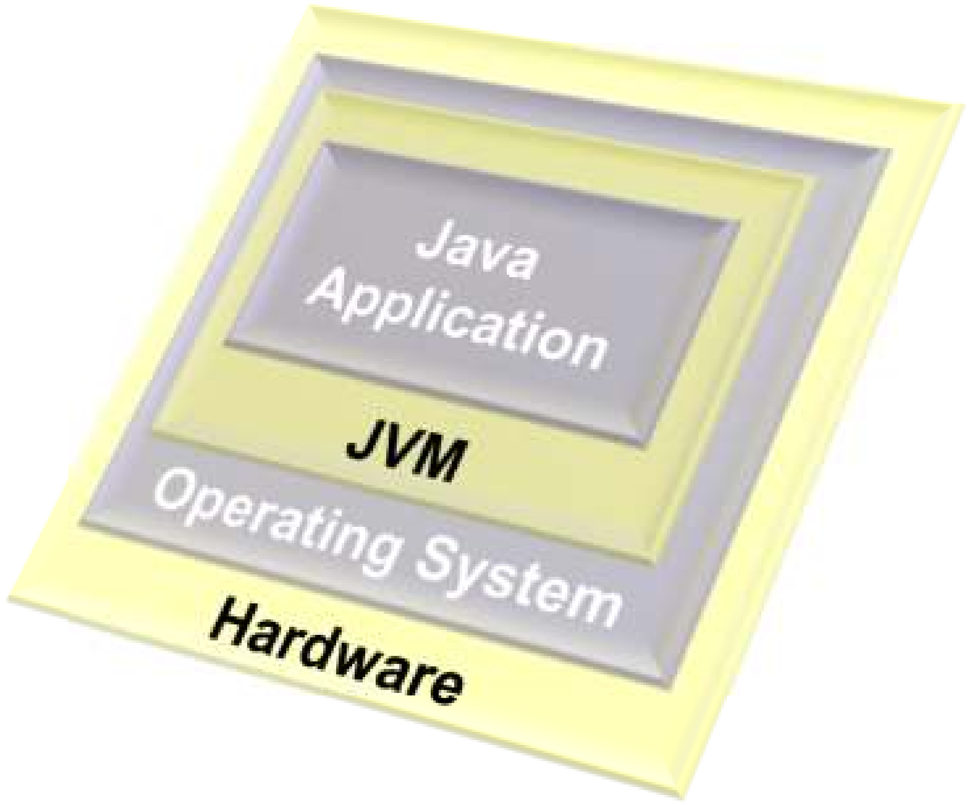
- Primary goals in the design of the Java programming language:
  - Simple, object oriented, and easy to learn.
  - Robust and secure.
  - Architecture neutral and portable.
  - Compiled and interpreted.
  - Execute with high performance and in network.
  - Threaded, and dynamic.

Java Features

- Java is easy to learn!
  - Syntax of C++
  - Dynamic Memory Management (Garbage Collection)
  - No pointers

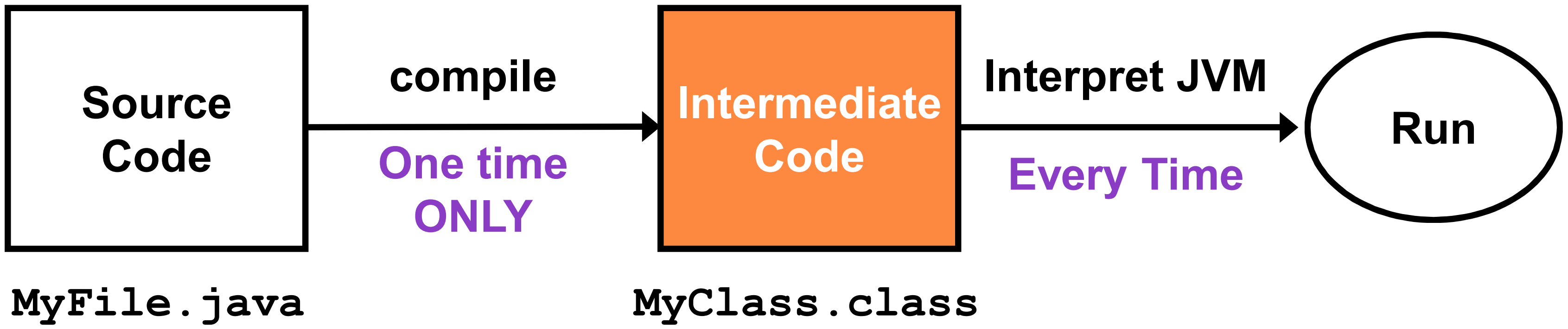
Java Features cont'd

- Machine and Platform Independent



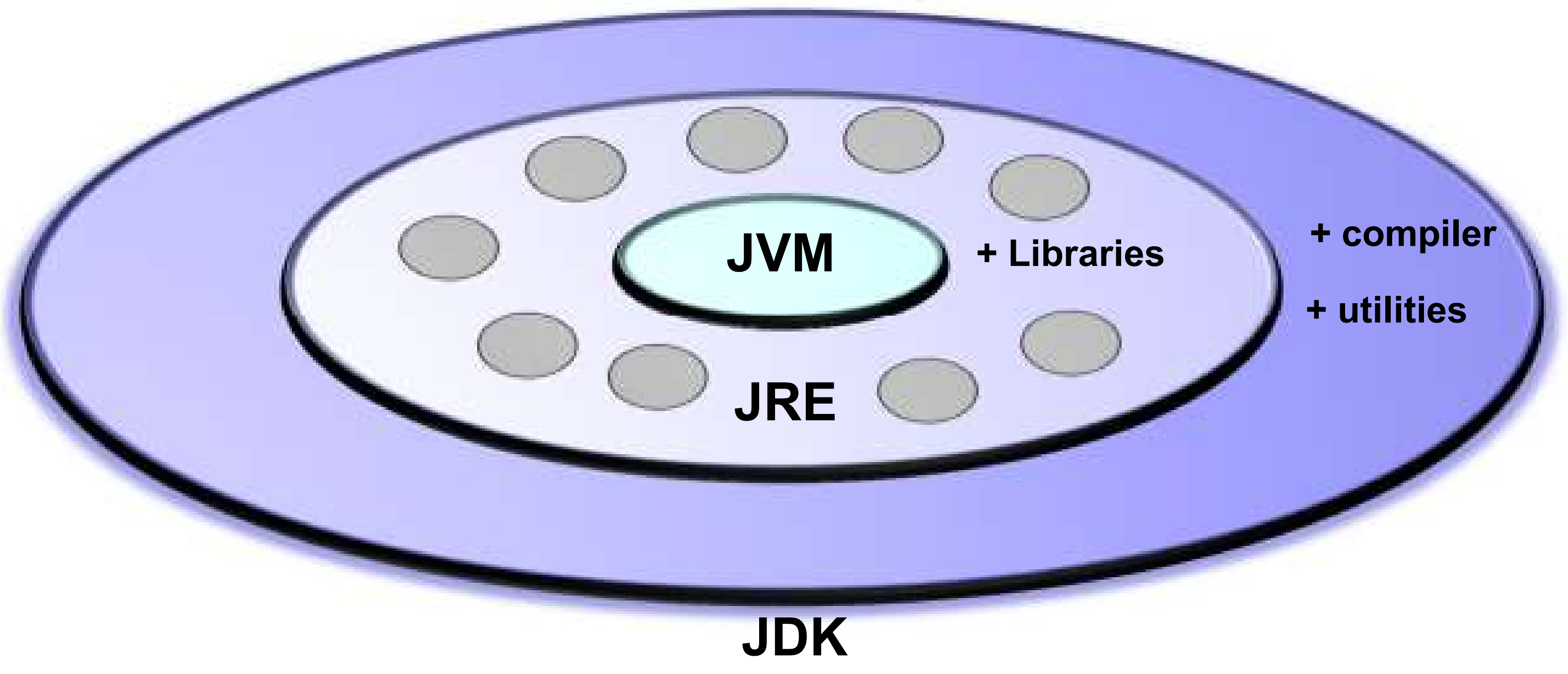
Java Features cont'd

- Java is both, compiled and interpreted



Java Features cont’d

- Java depends on dynamic linking of libraries



Java Features cont'd

- Java is fully Object Oriented
  - Made up of Classes.
  - No multiple Inheritance.
- Java is a multithreaded language
  - You can create programs that run multiple threads of execution in parallel.
    - Ex: GUI thread, Event Handling thread, GC thread
- Java is networked
  - Predefined classes are available to simplify network programming through Sockets(TCP-UDP)

Installing JDK

- Download the JDK:
  - If you use Solaris, Linux, Windows, or Mac point your browser to http://www.oracle.com/technetwork/java/javase/downloads/index.html to download the JDK.
  - Look for version 8.0, and pick your platform.
  - After downloading the JDK, follow the platform-dependent installation directions.

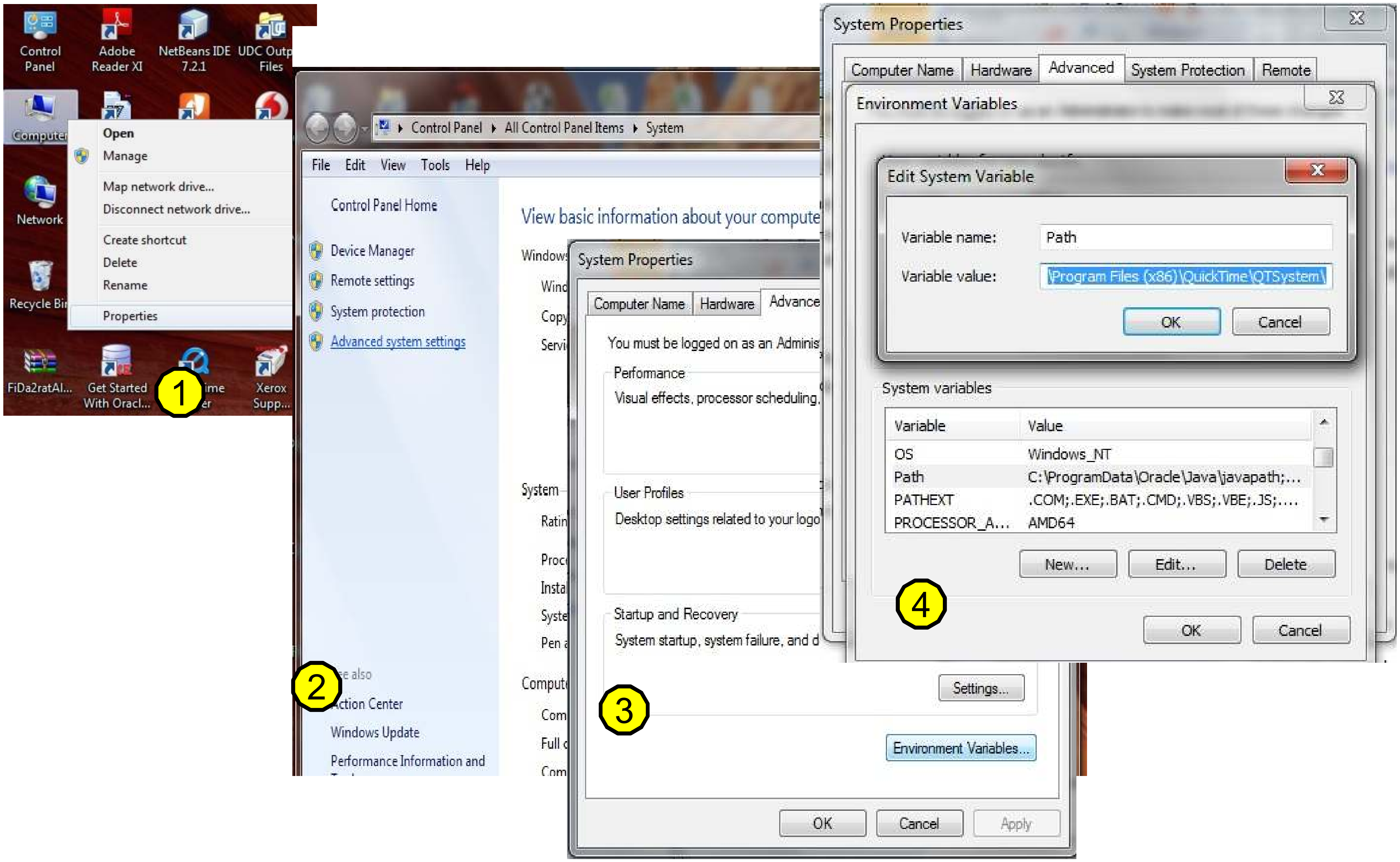
    https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Java Environment Setup “windows”

- Once you installed Java on your machine,
  - you would need to set environment variables to point to correct installation directories:
    - Assuming you have installed Java in

      c:\Program Files\java\jdk directory\bin\
    - Right-click on **'My Computer'** and select **'Properties'**.
    - Click on the **'Environment variables'** button under the **'Advanced'** tab.
    - Now alter the **'Path'** variable so that it also contains the path to the Java executable.

Java Environment Setup “windows”



Variable name: Path

Variable value: \Program Files (x86)\QuickTime\QTSystem\

Variable	Value
OS	Windows_NT
Path	C:\ProgramData\Oracle\Java\javapath;...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

Lesson 2

Basic Java Concepts

Programming Techniques Before OOP

- The Programming Techniques before OOP are:

  1. **Linear Programming:**

     - Maximum thing you can do is jumping “i.e. goto”, but the entire program is in order of lines.
     - Example: Fortran language
     - **Advantages:** suitable for small programs
     - **Disadvantages:**
       – Redundancy (Repetition)
       – Only one programmer can work at a time
       – Difficult to debug, difficult to maintain

Programming Techniques Before OOP

- The Programming Techniques before OOP are:

  2. **Structure Programming:**

     - Repeated code is organized in functions.
     - Functions are called where appropriate
     - Example: C language

     - **Advantages:** Allows the use of the teamwork.

     - **Disadvantages:**
       - All data is shared: **no protection**
       - More difficult to **modify**
       - Hard to manage **complexity data**

Introduction to OOP

- **What is OOP?**
  - OOP is mapping the real world to Software
  - OOP is a ***community*** of interacting agents called ***objects***.
  - Each object has a role to play.
  - Each object provides a ***service*** or performs an action that is used by other objects of the community.
  - Action is initiated by the transmission of a ***message*** to an object responsible for the actions.

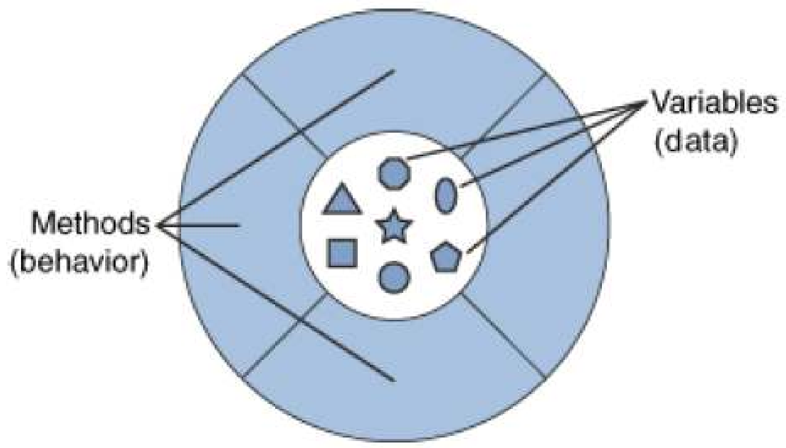
Introduction to OOP

- **What is OOP?**
  - All objects are instances of a ***class***.
  - The method invoked by an object is determined by the class of the receiver.
  - All objects of a given class use the same method in response to similar messages.

Introduction to OOP - Object

- **What is an Object?**
  - An object is a software bundle of variables and related methods.

- Object consist of:
  - **Data** (object's Attributes)
  - **Behavior** (object's methods)

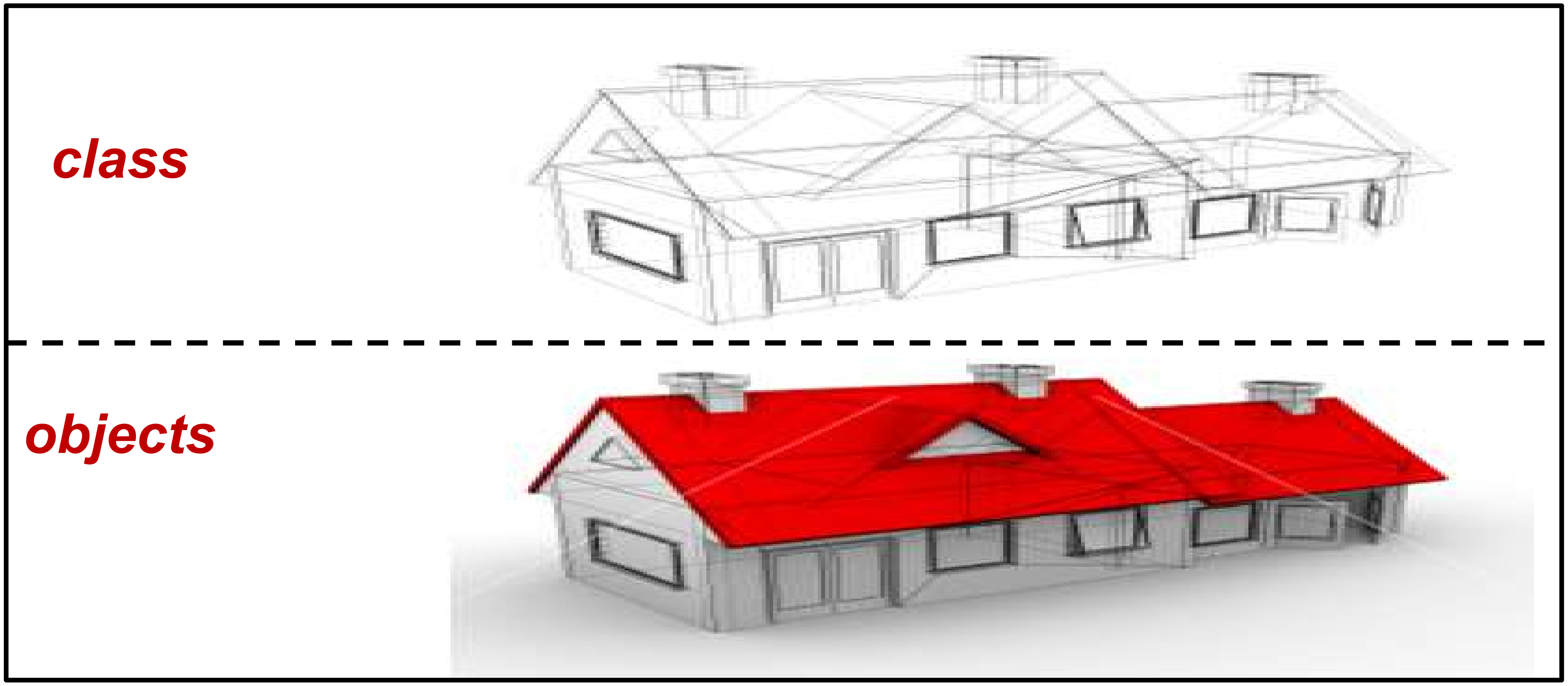


Introduction to OOP - Class

- **What is a Class?**
  - A class is a blueprint of objects.
  - A class is an object factory.
  - A class is the template to create the object.
  - A class is a user defined datatype
- **Object:**
  - An object is an instance of a class.
  - The property values of an object instance is different from the ones of other object instances of a same class
  - Object instances of the same class share the same behavior (methods).

Introduction to OOP – Object & Class

- *Class* reflects concepts.
- *Object* reflects instances that embody those concepts.



How to create a class?

- To define a class, we write:

```
<access-modifier>* class <name>
{
        <attributeDeclaration>*
        <constructorDeclaration>*
        <methodDeclaration>*
}
```

- Example:

```
class StudentRecord {
        //we'll add more code here later
}
```

Coding Guidelines

- Think of an appropriate name for your class.
  - Don't use XYZ or any random names.
- Class names starts with a CAPITAL letter.
  - not a requirement it is a convention

Declaring Properties (Attributes)

- declare a certain attribute for our class, we write,

```
<access-modifier>* <type> <name> [= <default_value>];
```

- Example:

```
class StudentRecord {
             // Instance variables
             public String       name;
             public String       address;
             private int         age    =     15;
             /*we'll add more code here later */
}
```

Declaring Properties (Attributes)

- **Access modifiers:**

1. **Public attributes:**
   - The access availability inside or outside the class.
2. **Private attributes:**
   - The access availability within the class only.

Declaring Methods

- declare a certain method for our class, we write,

```
<modifier>* <Return type> <name> ([<Param Type> <Param
Name>]*)
{
      <Statement>*
}
```

- Example:

```
class StudentRecord {
      private String name;
      public String getName(){ return name; }
      public void setName(String str){ name=str; }
      public static String getSchool(){..........}
}
```

Declaring Methods

- The following are characteristics of methods:
  - It can return one or no values
  - It may accept as many parameters it needs or no parameter at all.
    - Parameters are also called arguments.
  - After the method has finished execution, it goes back to the method that called it.
  - Method names should start with a small letter.
  - Method names should be verbs.

Declaring Properties (Methods)

- **Access modifiers:**

1. **Public method:**
   - The access availability inside or outside the class.
2. **Private method:**
   - The access availability within the class only.
3. **Static method:**
   - Methods that can be invoked without instantiating a class.
   - To call a static method, just type,

     `Classname.staticMethodName(params);`

Big Example

```
class Student{
   String firstName,lastName;
   int age;
   double mathScore;
   double scienceScore;
   int getAge(){ return age; }
   void setAge(int g){ age=g; }
   public static String getSchool(){//return school
   name}
   double average(){
        double avg=0;
        avg=(mathScore+scienceScore)/2;
        return avg;
   }
}
```

Create Object Instance

- To create an object instance of a class,
  - we use the **`new`** operator.
- For example,
  - if you want to create an instance of the class Student, we write the following code,

```
Student s1 = new Student();
```

- The new operator
  - Allocates a memory for that object and returns a reference of that memory location to you.
  - When you create an object, you actually invoke the class' constructor.

Accessing members of class

- To access members of class:

```
class Test {
      void testMethod(){
            Student s1 = new Student();
            s1.setAge(10);
            double d;
            d = s1.average();
            String s = Student.getSchool();
      }
}
```

First Java Application

```
class HelloWorld
{
  public static void main(String[] args)
   {
      System.out.println(“Hello Java”);
   }
}
```

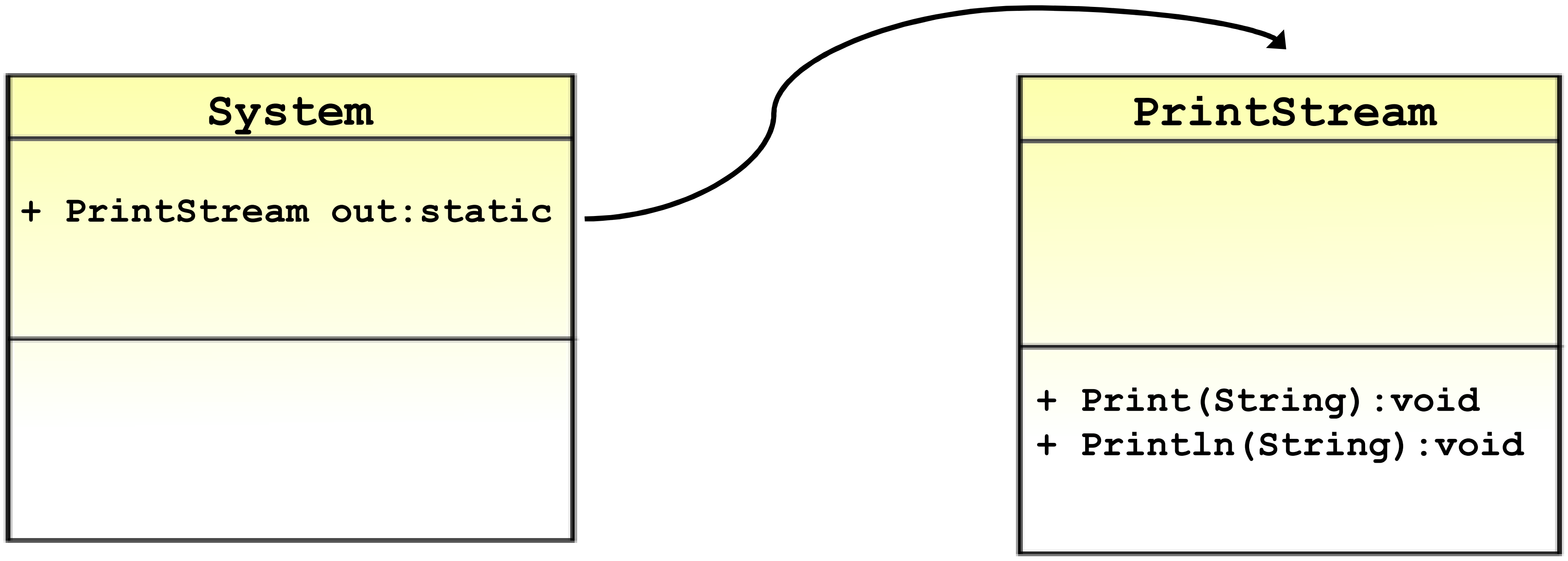
File name: `hello.java`

First Java Application cont'd

- The `main()` method:
  - Must return void.
  - Must be static.
    - because it is the first method that is called by the Interpreter (`HelloWorld.main(..)`) even before any object is created.
  - Must be public to be directly accessible.
  - It accepts an array of strings as parameter.
    - This is useful when the operating system passes any command arguments from the prompt to the application.

System.out.println("Hello");

- **out** is a static reference that has been created in class **System**.
- **out** refers to an object of class **PrintStream**. It is a ready-made stream that is attached to the standard output (i.e. the screen).



Standard Naming Convention
"The Hungarian Notation."

- Class names:

  `MyTestClass` , `RentalItem`

- Method names:

  `myExampleMethod()` , `getCustomerName()`

- Variables:

  `mySampleVariable` , `customerName`

- Constants:

  `MY_STATIC_VAR` , `MAX_NUMBER`

- Package:

  `pkg1` , `util` , `accesslayer`

Compiling and Running a Java Application

- **To compile:**

```
Prompt> javac hello.java
```

- If there are no compiler errors, then the file `HelloWorld.class` will be generated.
- **To run:**

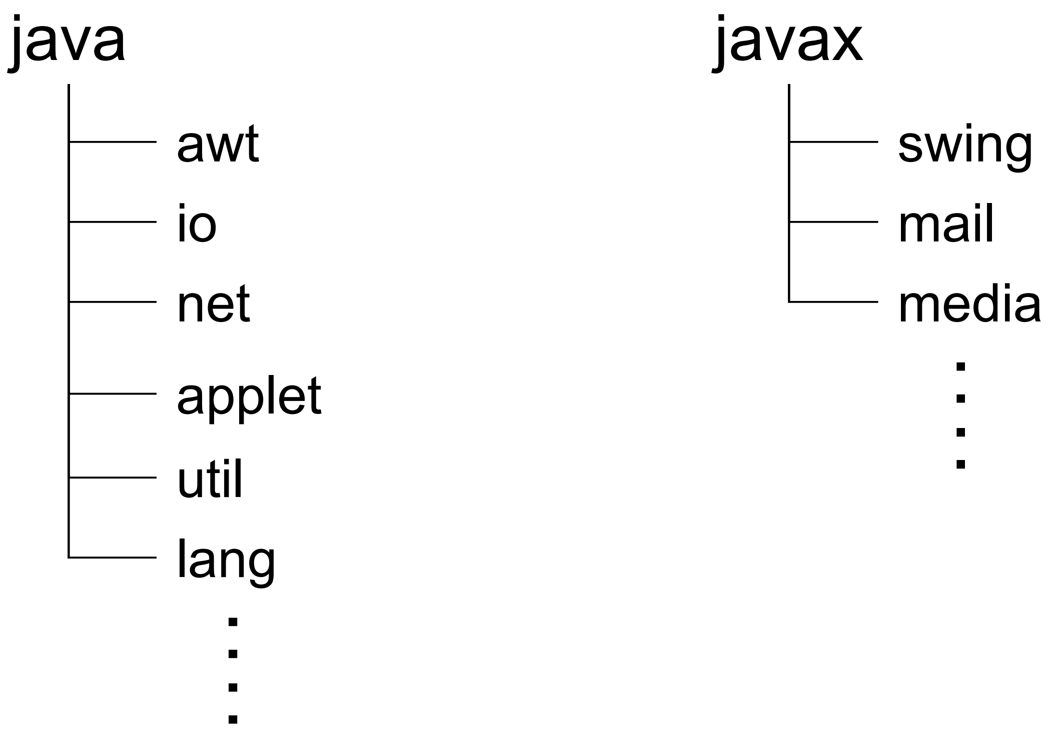
```
Prompt> java HelloWorld
Hello Java
Prompt>
```

Java Structure

- Classes are placed in packages.
- We must import any classes that we will use inside our application.
- Classes that exist in package `java.lang` are imported by default.
- Any Class by default extends `Object` class.

Java Structure cont'd

- The following are some package names that contain commonly used classes of the Java library:



Specifying a Package

- If no package is specified,
  - then the compiler places the .class file in the default package (i.e. the same folder of the .java file).
- To specify a package for your application,
  - write the following line of code at the beginning of your class:

```
package mypkg;
```

Specifying a Package

- To compile and place the .class in its proper location:

```
Prompt> javac -d . hello.java
```

Current Directory

- To run:

```
Prompt> java mypkg.HelloWorld
```

JAR File

- Packages can be brought together in one compressed JAR file.
- The classes of Java Runtime Libraries (JRE) exist in **`rt.jar`**.
- JAR files can be made executable by writing a certain property inside the **manifest.mf file** that points to the class that holds the **`main(..)`** method.

How to make JAR file

- To create a compressed JAR file:

```
prompt> jar cf <archive_name.jar> <files>
```

- **Example:**

```
prompt> jar cf App.jar HelloWorld.class
```

How to make JAR file cont'd

- To create an executable JAR file:
  1. Create text file that list the main class.
     "The class that has the main method"
  2. Write inside the text file this text:
     Main-Class: <class name>
  3. Then run the jar utility with this command line:

```
prompt>jar cmf <text-file> <archive_name.jar>
   <files>
```

Lab Assignments

1. Simple Prompt Application

- Create a simple non-GUI Application that prints out the following text on the command prompt:

  Hello Java

- **Note:** specify package and create executable jar file.
- **Bonus:** Modify the program to print a string that is passed as an argument from the command prompt.

2. Simple Prompt Application

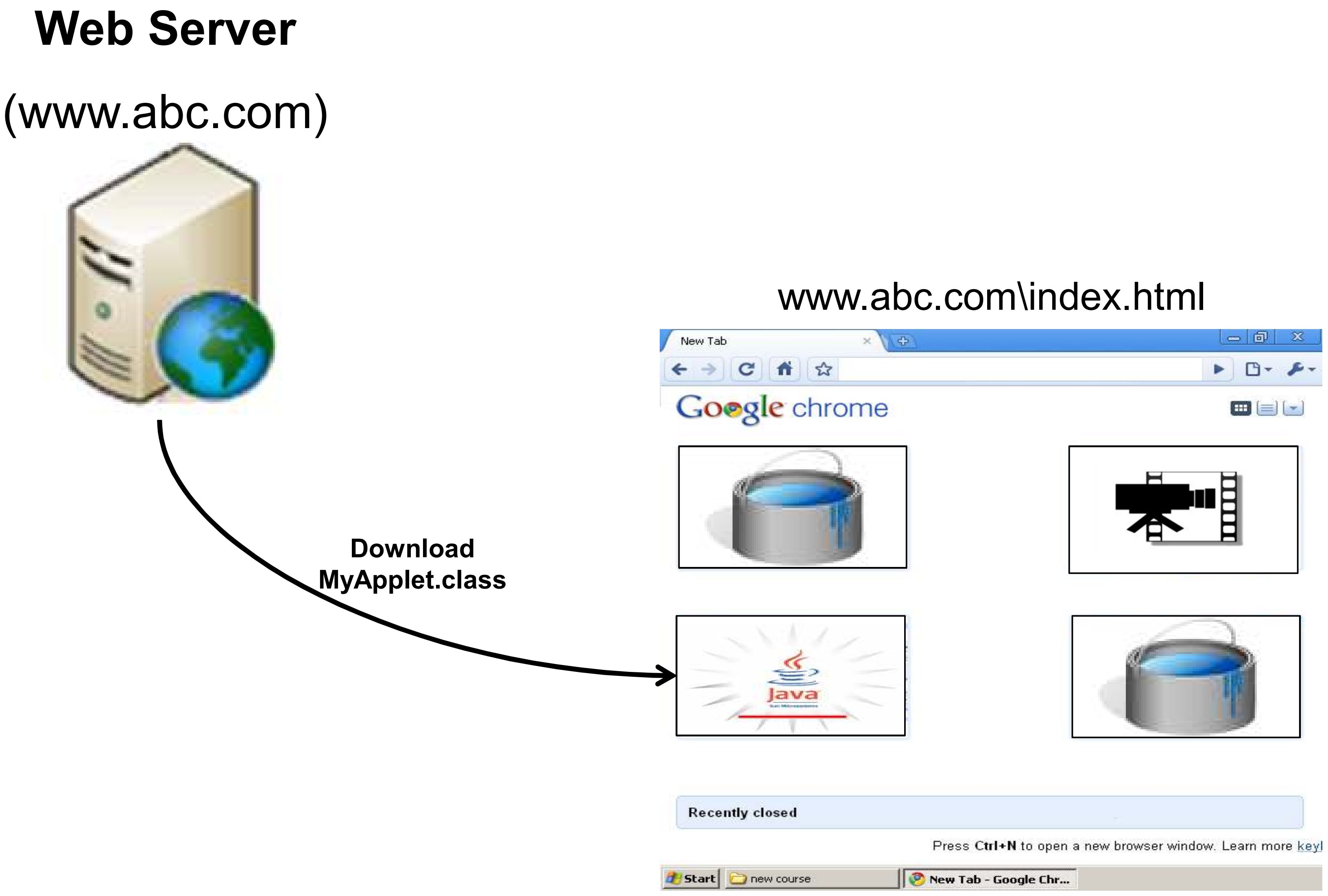
- Create a simple non-GUI Application that represent complex number and has two methods to add and subtract complex numbers:

Complex number: x + yi , 5+6i

Lesson 3

Applet

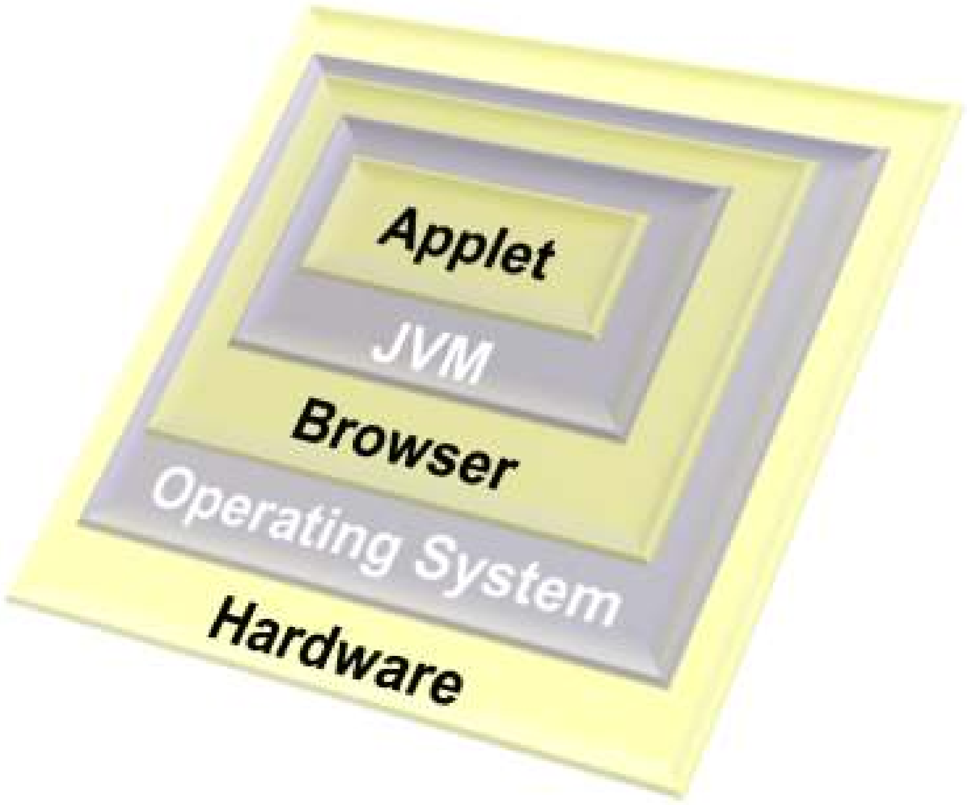
Overview



Web Server
(www.abc.com)
www.abc.com\index.html
Download MyApplet.class
New Tab
Google chrome
Java
Recently closed
Press Ctrl+N to open a new browser window. Learn more keyl
Start
new course
New Tab - Google Chr...

Applet Features

- Machine and Platform Independent



Applets

- An Applet is a client side Java program that runs inside the web browser.

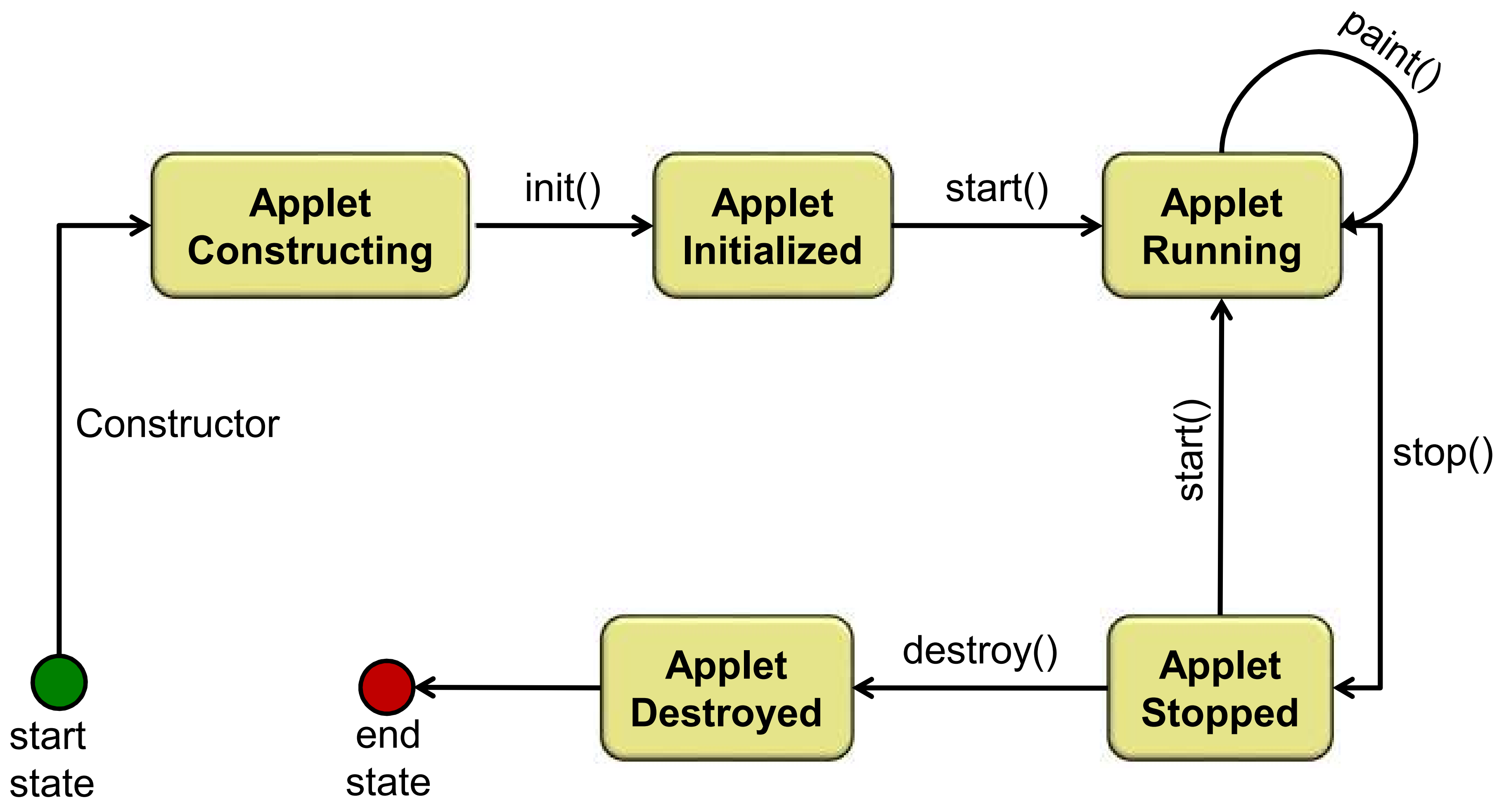
- The .class file of the applet is downloaded from the web server to the client's machine

- The JVM interprets and runs the applet inside the browser.

Applet Security

- In order to protect the client from malformed files or malicious code, the JVM enforce some security restrictions on the applet:
  - Syntax is checked before running.
  - No I/O operations on the hard disk.
  - Communicates only with the server from which it was downloaded.
- Applets can prompt the client for additional security privileges if needed.

Applet Life Cycle



Applet Life Cycle

- **The life cycle of Applet:**

- `init()`:
  - called when the applet is being initialized for the first time.

- `start()`:
  - called whenever the browser's window is activated.

- `paint(Graphics g)`:
  - called after **`start()`** to paint the applet, or
  - whenever the applet is repainted.

- `stop()`:
  - called whenever the browser's window is deactivated.

- `destroy()`:
  - called when the browser's window is closed.

Applet Life Cycle cont’d

- You can refresh the applet anytime by calling: `repaint()`,
  - which will invoke `update(Graphics g)` to clear the applet,
  - which in turn invokes `paint(Graphics g)` to draw the applet again.
- To create your own applet, you write a class that extends class `Applet`,
  - then you override the appropriate methods of the life cycle.

Basic Java Applet

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloApplet extends Applet{
  public void paint(Graphics g){
     g.drawString("Hello Java", 50, 100);
  }
}
```

Note: Your class must be made public or else the browser will not be able to access the class and create an object of it.

Basic Java Applet cont'd

- In order to run the applet we have to create a simple HTML web page, then we invoke the applet using the <applet> tag.
- The **`<applet>`** tag requires 3 mandatory attributes:
  - code
  - width
  - height
- An optional attribute is codebase, which specifies the path of the applet's package.

Basic Java Applet cont'd

- Write the following in an HTML file e.g. mypage.html:

```
<html>
  <body>
  <applet code="HelloApplet"
          width="400"  height="350">
</applet>
  </body>
</html>
```

Compiling and Running an Applet

- Save the Hello Applet Program in your assignments folder in a file named: `HelloApplet.java`
  - When a class is made public, then you have to name the file after it.
- To compile write in cmd this command:

```
javac HelloApplet.java
```

- An applet is not run like an application.
- Instead, you browse the HTML file from your web browser, or by using the applet viewer:

```
appletviewer mypage.html
```

from the command prompt.

Lab Exercise

1. Basic Applet

- Create an applet that displays: **Hello Java.**

- **Bonus:** Try to pass some parameters from the HTML page to the applet. For example, display the parameters on the applet.

  Hint:

  use the self closing tag: `<param   name=   value=   />`

Lesson 4

Data Types & Operators

Identifiers

- An identifier is the name given to a feature (variable, method, or class).
- An identifier can begin with either:
  - a letter,
  - $, or
  - underscore.
- Subsequent characters may be:
  - a letter,
  - $,
  - underscore, or
  - digits.

Data types

- Data types can be classified into two types:

Primitive

Boolean	`boolean`	1 bit	(true/false)
Integer	`byte`	1 B	$(-2^{7} \rightarrow 2^{7}-1)$ (-128 → +127)
	`short`	2 B	$(-2^{15} \rightarrow 2^{15}-1)$ (-32,768 to +32,767)
	`int`	4 B	$(-2^{31} \rightarrow 2^{31}-1)$
	`long`	8 B	$(-2^{63} \rightarrow 2^{63}-1)$
Floating Point	`float`	4 B	Standard: IEEE 754 Specification
	`double`	8 B	Standard: IEEE 754 Specification
Character	`char`	2 B	unsigned Unicode chars $(0 \rightarrow 2^{16}-1)$

Reference

Arrays
Classes
Interfaces

Wrapper Classes

- Each primitive data type has a corresponding wrapper class.

boolean	→	Boolean
byte	→	Byte
char	→	Character
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double

Wrapper Classes cont'd

- There are three reasons that you might use a wrapper class rather than a primitive:

  1. As an argument of a method that expects an object.
  2. To use constants defined by the class,
     - such as **MIN_VALUE** and **MAX_VALUE**, that provide the upper and lower bounds of the data type.
  3. To use class methods for
     - converting values to and from other primitive types,
     - converting to and from strings,
     - converting between number systems (decimal, octal, hexadecimal, binary).

Wrapper Classes cont'd

- They have useful methods that perform some general operation, for example:

`primitive xxxValue()`	→	convert wrapper object to primitive
`primitive parseXXX(String)`	→	convert String to primitive
`Wrapper valueOf(String)`	→	convert String to Wrapper

```
Integer i2 = new Integer(42);
byte b = i2.byteValue();
double d = i2.doubleValue();
```

```
String s3 = Integer.toHexString(254);
System.out.println("254 is " + s3);
```

Wrapper Classes cont'd

- They have special static representations, for example:

POSITIVE_INFINITY		In class Float & Double
NEGATIVE_INFINITY		
NaN	Not a Number	

Literals

- A literal is any value that can be assigned to a primitive data type or String.

boolean	true false	
char	‘a’ …. ’z’ ‘A’ …. ‘Z’	
	‘\u0000’ …. ‘\uFFFF’	
	‘\n’ ‘\r’ ‘\t’	
Integral data type	15	Decimal (int)
	15L	Decimal (long)
	017	Octal
	0XF	Hexadecimal
Floating point data type	73.8	double
	73.8F	float
	5.4 E-70	$5.4 * 10^{-70}$
	5.4 e+70	$5.4 * 10^{70}$

Literals “Java 7”

- In Java SE 7,
  - Any number of underscore characters (_) can appear anywhere between digits in a numerical literal.
  - This feature enables you, to separate groups of digits in numeric literals, which can improve the readability.
    - similar to how you would use a comma as a separator.
- Examples:
  - `long creditCardNumber = 1234_5678_9012_3456L;`
  - `long socialSecurityNumber = 999_99_9999L;`
  - `float pi = 3.14_15F;`
  - `long hexBytes = 0xFF_EC_DE_5E;`
  - `byte nybbles = 0b0010_0101;`
  - `long bytes = 0b11010010_01101001_10010100_10010010;`

Reference Data types: Classes

- General syntax for creating an object:

```
MyClass myRef;                 // just a reference
myRef = new MyClass();         // construct a new object
```

- Or on one line:

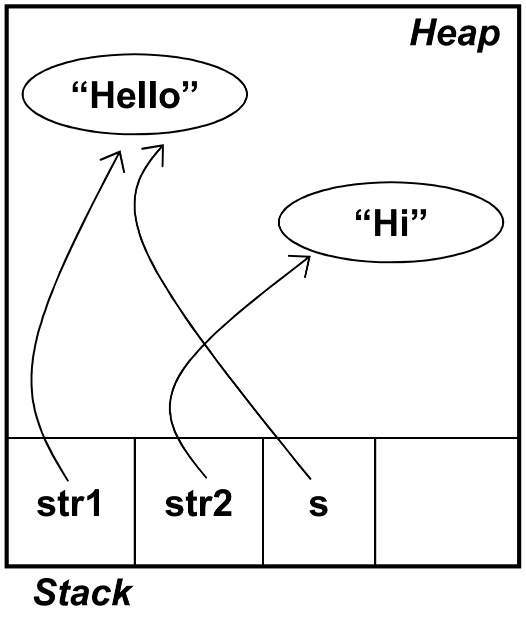
```
MyClass myRef = new MyClass();
```

- An object is garbage collected when there is no reference pointing to it.

Reference Data types: Classes cont’d

```
String str1;                        // just a null reference
str1 = new String(“Hello”);  // object construction

String str2 = new String(“Hi”);

String s = str1;       //two references to the same object

str1 = null;

s = null;        // The object containing “Hello” will
                 // now be eligible for garbage collection.



str1.anyMethod();     // ILLEGAL!
                      //Throws NullPointerException
```

Memory



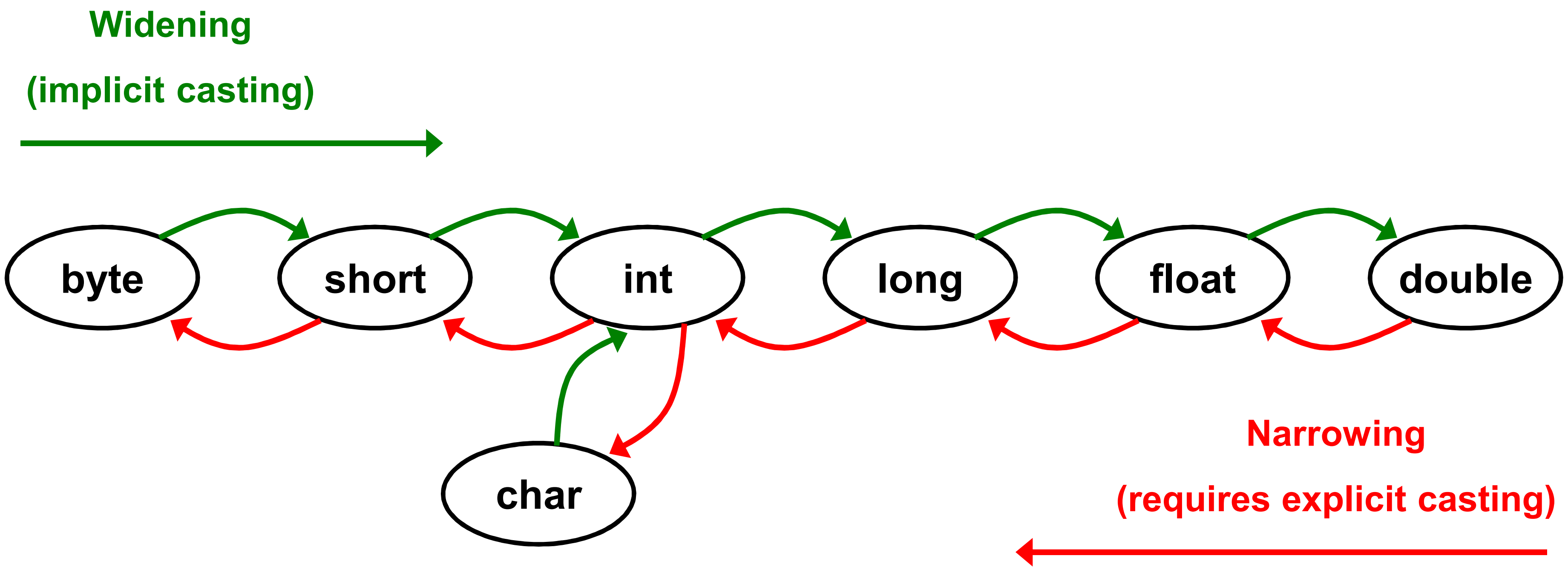
Operators

- Operators are classified into the following categories:
  - Unary Operators.
  - Arithmetic Operators.
  - Assignment Operators.
  - Relational Operators.
  - Shift Operators.
  - Bitwise and Logical Operators.
  - Short Circuit Operators.
  - Ternary Operator.

Operators cont'd

- Unary Operators:

+	-	++	--	!	~	( )
positive	negative	increment	decrement	boolean complement	bitwise inversion	casting



Operators cont'd

- Arithmetic Operators:

+	-	*	/	%
add	subtract	multiply	division	modulo

- Assignment Operators:

=	+=	-=	*=	/=	%=	&=	|=	^=

- Relational Operators:

<	<=	>	>=	==	!=	instanceof

Operations must be performed on homogeneous data types

Operators cont'd

```
byte b=10;
byte b1=15;
byte b2=b+b1;
```

Value of b2 is ?

↓

Compilation Error – Explicit Cast Needed to convert from integer to byte

5 % 2	→ 1
5 % -2	→ 1
-5 % 2	→ -1
-5 % -2	→ -1

```
int x = 1234567899
int y = 567899999
int z = x*y/x
```

z = ?

↓

Unexpected results

Operators cont'd

- Shift Operators:

>>	<<	>>>
right shift	left shift	unsigned right shift

- Bitwise and Logical Operators:

&	|	^
AND	OR	XOR

- Short Circuit Operators:

&&	||
(condition1 AND condition2)	(condition1 OR condition2)

Operators cont'd

- Ternary Operator:

```
condition ?true statement:false statement
```

```
int y = 15;                          If(y<z)
int z = 12;                 →              x=10;
int x = y<z? 10 : 11;               else
                                           x=11;
```

Operators cont'd

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
Bitwise and Logical AND	&
bitwise exclusive OR	^
Bitwise and Logical inclusive OR	|
Short Circuit AND	&&
Short Circuit OR	||
ternary	? :
assignment	= op=

Lesson 5

Using Arrays & Strings

What is Array?

- An Array is a collection of variables of the same data type.
- Each element can hold a single item.
- Items can be primitives or object references.
- The length of the array is determined when it is created.

What is Array?

- Java Arrays are homogeneous.
- You can create:
  - An array of primitives,
  - An array of object references, or
  - An array of arrays.
- If you create an array of object references, then you can store subtypes of the declared type.

Declaring an Array

- General syntax for creating an array:

```
Datatype[]  arrayIdentifier;               // Declaration
arrayIdentifier = new Datatype [size];  //
   Construction
```

- Or on one line, hard coded values:

```
Datatype[] arrayIdentifier = { val1, val2, val3,
   val4 };
```

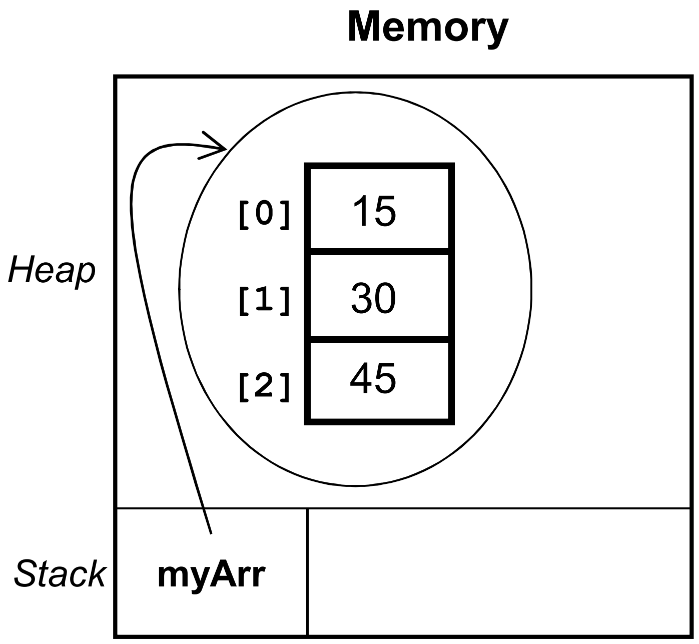
- To determine the size (number of elements) of an array at runtime, use:

```
                    arrayIdentifier.length
```

Declaring an Array cont'd

- **Example1:** Array of Primitives:

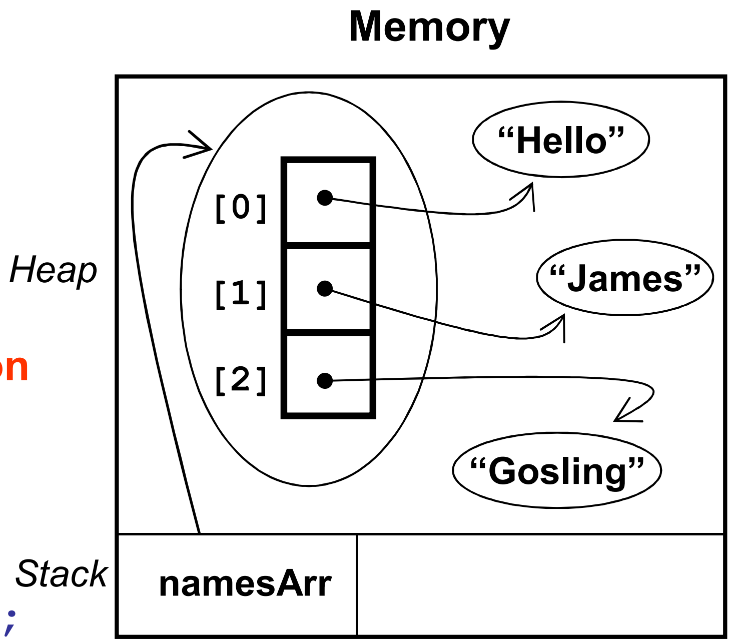
```
int[] myArr;

myArr = new int[3];

myArr[0] = 15 ;
myArr[1] = 30 ;
myArr[2] = 45 ;

System.out.println(myArr[2]);


myArr[3] = … ; // ILLEGAL!
         //Throws ArrayIndexOutOfBoundsException
```



Declaring an Array cont'd

- **Example2:** Array of Object References:

```
String[] namesArr;

namesArr = new String[3];

namesArr[0].anyMethod()  // ILLEGAL!
                         //Throws NullPointerException

namesArr[0] = new String("Hello");
namesArr[1] = new String("James");
namesArr[2] = new String("Gosling");

System.out.println(namesArr[1]);
```



String Operations

- Although String is a reference data type (class),
  - it may figuratively be considered as the 9th data type because of its special syntax and operations.
  - Creating String Object:

```
String myStr1 = new String(“Welcome”);
String sp1 = “Welcome”;
String sp2 = “ to Java”;
```

  - Testing for String equality:

```
if(myStr1.equals(sp1))

if(myStr1.equalsIgnoreCase(sp1))

if(myStr1 == sp1)
 // Shallow Comparison (just compares the references)
```

Strings Operations cont'd

- The '+' and '+=' operators were overloaded for class String to be used in concatenation.

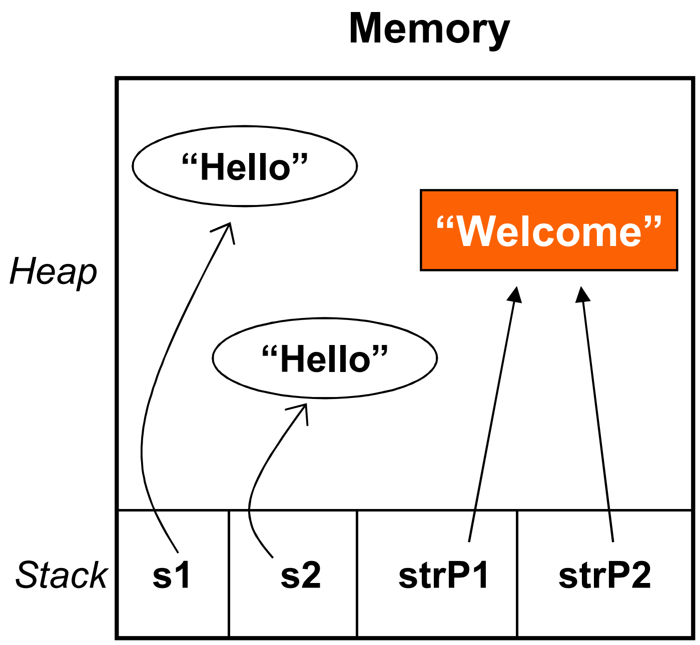
```
String str = myStr1 + sp2;          // "Welcome to Java"
str += " Programming";              // "Welcome to Java Programming"
str = str.concat(" Language");      // "Welcome to Java Programming Language"
```

- Objects of class String are immutable
  - you can't modify the contents of a String object after construction.
- Concatenation Operations always return a new String object that holds the result of the concatenation. The original objects remain unchanged.

String Pool

- String objects that are created without using the “new” keyword are said to belong to the “String Pool”.

```
String s1 = new String(“Hello”);
String s2 = new String(“Hello”);

String strP1 = “Welcome” ;
String strP2 = “Welcome” ;
```



String Pool cont'd

- String objects in the pool have a special behavior:
  - If we attempt to create a fresh String object with exactly the same characters as an object that already exists in the pool (case sensitive), then no new object will be created.
  - Instead, the newly declared reference will point to the existing object in the pool.
- Such behavior results in a better performance and saves some heap memory.
- Remember: objects of class String are immutable.

Lesson 6

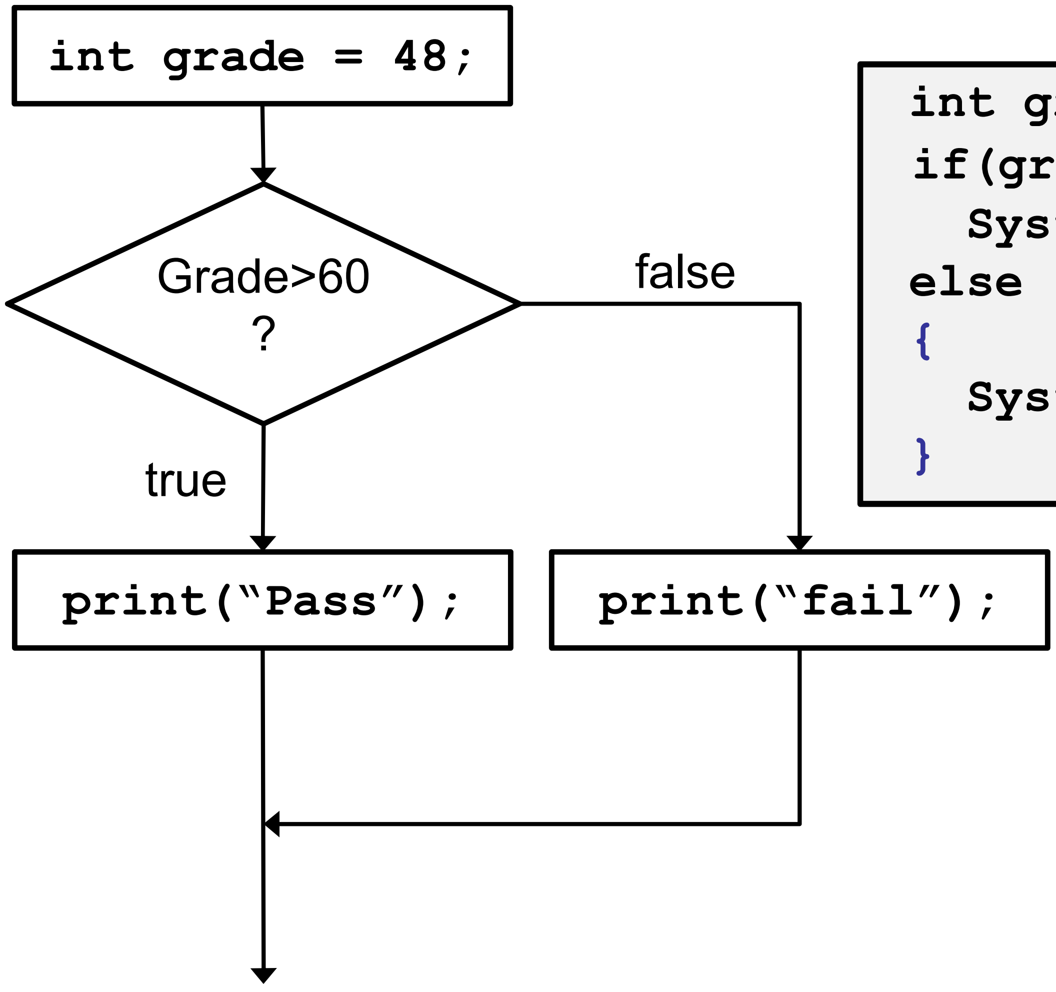
Controlling Program Flow

Flow Control: Branching - if, else

- The if and else blocks are used for binary branching.
- **Syntax:**

```
if(boolean_expr)
{
        …
        …    //true statements
        …
}
[else]
{
        …
        …    //false statements
        …
}
```

if, else Example



```
int grade = 48;
if(grade > 60)
  System.out.println("Pass");
else
{
  System.out.println("Fail");
}
```

Flow Control: Branching - switch

- The switch block is used for multiple branching.
- **Syntax:**

```
switch(myVariable){
        case value1:
           …
           …
        break;
        case value2:
           …
           …
        break;
        default:
           …
}
```

- byte
- short
- int
- char
- enum
- String "Java 7"

Flow Control: Branching – switch (EX.)

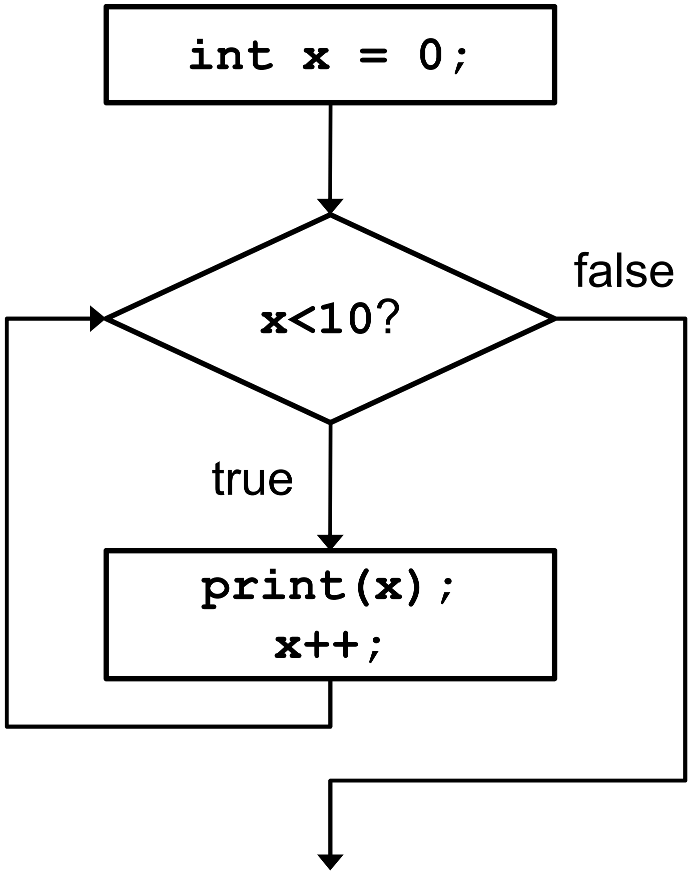
```
public class StringSwitchDemo {
  public int getMonthNumber(String month) {
    int monthNumber = 0;
    switch (month.toLowerCase()) {
      case "january": monthNumber = 1; break;
      case "february": monthNumber = 2; break;
      case "march": monthNumber = 3; break;
      ........
      default: monthNumber = 0; break;
    }
    return monthNumber;
  }
}
```

Flow Control: Iteration – while loop

- The while loop is used when the termination condition occurs unexpectedly and is checked at the beginning.
- **Syntax**:

```
while (boolean_condition)
{
        …
        …
        …
}
```

while loop Example



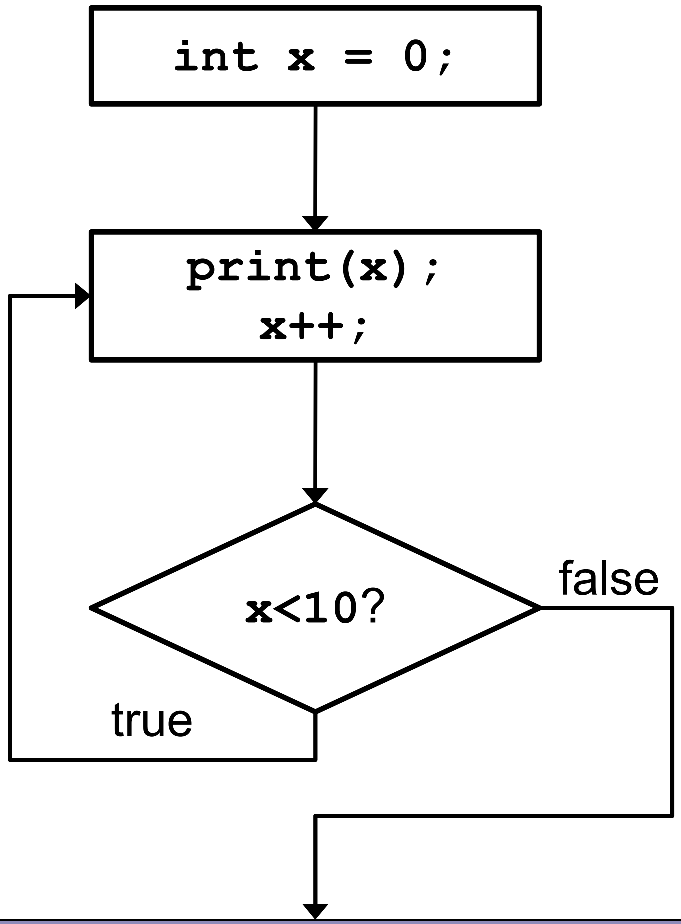
```
int x = 0;
while (x<10) {
    System.out.println(x);
    x++;
}
```

Flow Control: Iteration – do..while loop

- The do..while loop is used when the termination condition occurs unexpectedly and is checked at the end.
- **Syntax:**

```
do
{
        …
        …
        …
}
while(boolean_condition);
```

do..while loop Example



```
int x = 0;
do{
    System.out.println(x);
    x++;
} while (x<10);
```

Flow Control: Iteration – for loop

- The for loop is used when the number of iterations is predetermined.
- **Syntax:**

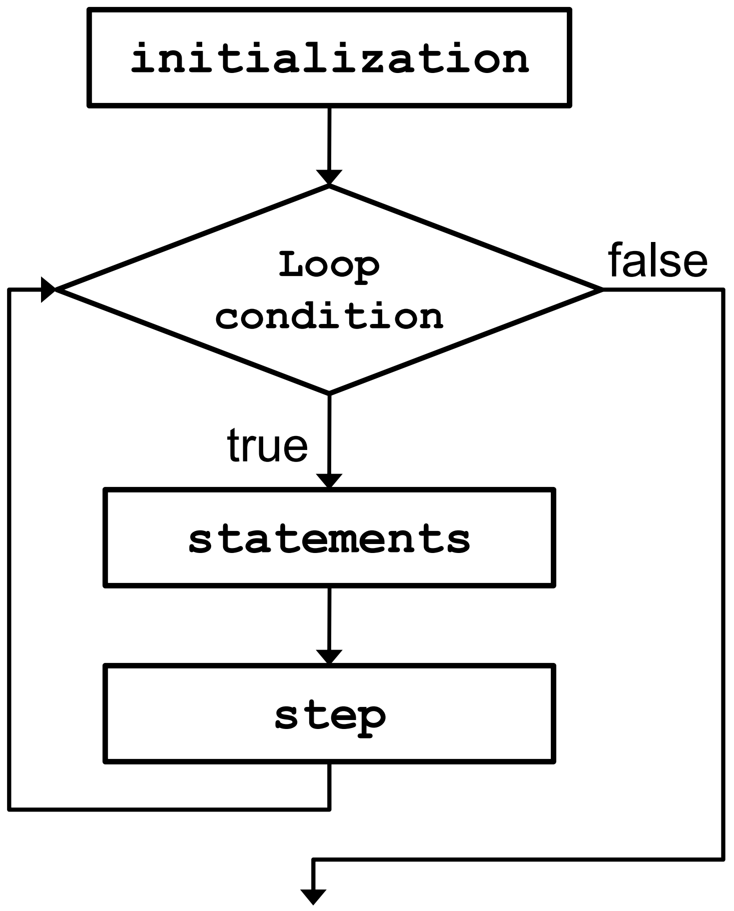
```
for (initialization ; loop_condition ; step)
{
        …
        …
        …
}
```

```
for (int i=0 ; i<10 ; i++)
{
        …
        …
}
```

- You may use the `break` and `continue` keywords to skip or terminate the iterations.

Flow Control: Iteration – for loop

```
for (initialization ; loop_condition ; step)
```



Flow Control: Iteration –Enhanced for loop

```
for (type identifier : iterable_expression)
{
        // statements
}
```

- **The first element:**
  - is an identifier of the same type as the iterable_expression
- **The second element:**
  - is an expression specifying a collection of objects or values of the specified type.
- The enhanced loop is used when we want to iterate over arrays or collections.

Flow Control: Iteration –Enhanced for loop example

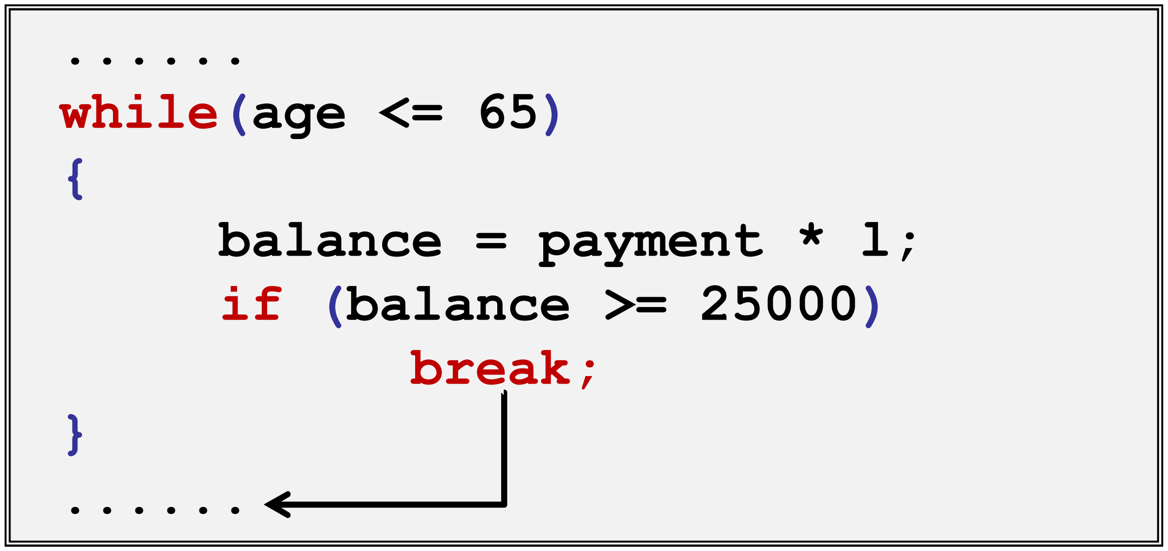
```
double[] samples = new double[50];
```

```
double average = 0.0;
for(int i=0;i<samples.length;i++)
{
     average += samples[i];
}

average /= samples.length;
```

```
double average = 0.0;
for(double value : samples)
{
     average += value;
}
average /= samples.length;
```

The `break` statement

- The break statement can be used in loops or switch.
- It transfers control to the first statement after the loop body or switch body.



```
......
while(age <= 65)
{
     balance = payment * 1;
     if (balance >= 25000)
           break;
}
......
```

The `continue` statement

- The continue statement can be used Only in loops.
- Abandons the current loop iteration and jumps to the next loop iteration.

```
......
for(int year=2000; year<= 2099; year++){
     if (year % 100 == 0)
           continue;
}
......
```

Comments in Java

- To comment a single line:

```
// write a comment here
```

- To comment multiple lines:

```
/*   comment line 1
     comment line 2
      comment line 3 */
```

Printable Class Documentation (Javadoc)

- To write professional class, method, or variable documentation:

```
/** javadoc line 1
    javadoc line 2
    javadoc line 3 */
```

  - You can then produce the HTML output by typing the following command at the command prompt:

```
javadoc myfile.java
```

Printable Class Documentation (Javadoc)

- The `Javadoc` tool parses tags within a Java doc comment.
- These doc tags enable you to
  - auto generate a complete, well-formatted API documentation from your source code.
- The tags start with (@).
- A tag must start at the beginning of a line.

Example of Javadoc

- Example 1:

```
/**
 * @author khaled
 */
```

- Example 2:

```
/**
* @param args the command line arguments
*/
```

Lab Exercise

1. Command Line Calculator

- Create a simple non-GUI Application that carries out the functionality of a basic calculator (addition, subtraction, multiplication, and division).

- The program, for example, should be run by typing the following at the command prompt:

  java Calc 70 + 30

2. String Separator

- Create a non-GUI Application that accepts a well formed IP Address in the form of a string and cuts it into separate parts based on the dot delimiter.

- The program, for example, should be run by typing the following at the command prompt:

  java IPCutter 163.121.12.30

- The output should then be:

```
163
121
12
30
```

- Write a program that print the following patterns:

1.

```
*
**
***
****
*****
******
```

2.

```
     ★
    ★ ★
   ★ ★ ★
  ★ ★ ★ ★
 ★ ★ ★ ★ ★
★ ★ ★ ★ ★ ★
```

Lesson 7

Modifiers-Access Specifiers Essential Java Classes(Graphics-Font-Color-Exception Classes)

Lesson 7 Outline

- ❑ Modifiers and Access Specifiers
- ❑ Graphics, Color, and Font Class
- ❑ Exception Handling

Modifiers and Access Specifiers

- Modifiers and Access Specifiers are a set of keywords that affect the way we work with features (classes, methods, and variables).

- The following table illustrates these keywords and how they are used.

Modifiers and Access Specifiers cont'd

Keyword	Top Level Class	Methods	Variables	Free Floating Block
public	Yes	Yes	Yes	-
protected	-	Yes	Yes	-
(friendly)*	Yes	Yes	Yes	-
private	-	Yes	Yes	-
final	Yes	Yes	Yes	-
static	-	Yes	Yes	Yes
abstract	Yes	Yes	-	-
native	-	Yes	-	-
transient	-	-	Yes	-
volatile	-	-	Yes	-
synchronized	-	Yes	-	-

Graphics Class

- The Graphics object is your means of communication with the graphics display.
- You can use it to draw strings, basic shapes, and show images.
- You can also use it to specify the color and font you want.
- You can write a string using the following method:

```
void drawString(String str, int x, int y)
```

Graphics Class cont'd

- Some basic shapes can be drawn using the following methods:

```
void drawLine(int x1, int y1, int x2, int y2);

void drawRect(int x, int y, int width, int height);
void fillRect(int x, int y, int width, int height);

void drawOval(int x, int y, int width, int height);
void fillOval(int x, int y, int width, int height);

void drawArc (int x, int y, int width, int height,
              int startAngle, int arcAngle)
void fillArc (int x, int y, int width, int height,
              int startAngle, int arcAngle)
```

Color Class

- In order to work with colors in your GUI application you use the Color class.

- Commonly Used Constructor(s):
  - `Color(int r, int g, int b)`
  - `Color(float r, float g, float b)`

- Commonly Used Method(s):
  - `int getRed()`
  - `int getGreen()`
  - `int getBlue()`
  - `Color darker()`
  - `Color brighter()`

- Objects of class Color are immutable.

Color Class cont’d

- There are 13 predefined color objects in Java. They are all declared as **`public static final`** objects in class Color itself:
  - **`Color.RED`**
  - **`Color.ORANGE`**
  - **`Color.PINK`**
  - **`Color.YELLOW`**
  - **`Color.GREEN`**
  - **`Color.BLUE`**
  - **`Color.CYAN`**
  - **`Color.MAGENTA`**
  - **`Color.GRAY`**
  - **`Color.DARK_GRAY`**
  - **`Color.LIGHT_GRAY`**
  - **`Color.WHITE`**
  - **`Color.BLACK`**

Color Class cont’d

- To specify a certain color to be used when drawing on the applet’s Graphics object use the following method of class Graphics:
  - `void setColor (Color c)`

- To change the colors of the foregoround or background of any Component, use the following method of class Component:
  - `void setForeground(Color c)`
  - `Void setBackground(Color c)`

Font Class

- In order to create and specify fonts in your GUI application you use the Font class.

- Commonly Used Constructor(s):
  - `Font(String name, int style, int size)`

- To specify a certain font to be used when drawing on the applet's Graphics object use the following method of class Graphics:
  - `void setFont (Font f)`

- To change the font of any Component, use the following method of class Component:
  - `void setFont(Font f)`

Font Class cont’d

- To obtain the list of basic fonts supported by all platforms you can write the following line of code:

```
String[] s = Toolkit.getDefaultToolkit().getFontList();
```

- Objects of class Font are immutable.

Exceptions

- What are Exceptions?
- Exception handling
- The try-catch statement
- The finally clause
- Exception propagation
- Exception classes hierarchy
- Exception types

Exceptions

- An *exception* is an object that describes an unusual or incorrect situation
- Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program
- A program can be separated into a normal execution flow and an *exception execution flow*
- An *error* is also represented as an object in Java, but usually represents an unrecoverable situation and should not be caught

Exception Handling

- The Java API has a predefined set of exceptions that can occur during execution
- A program can handle with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it in another place in the program
- The manner in which an exception is processed is an important design consideration

Exception Handling

- If an exception is ignored (not caught) by the program, the program will terminate and produce an appropriate message
- The message includes a *call stack trace* that:
  - indicates the line on which the exception occurred
  - shows the method call trail that lead to the attempted execution of the offending line

Example 1

```
public class Example1 {
 public static void main(String[] args) throws AnException
 {
      Example1 ref=new Example1();
      ref.myMethod(5);
 }
 public void myMethod(int x) throws AnException
 {
   if (x>0) throw new AnException("Exception in method");
 }
}
```

The try Statement

- To handle an exception in a program, use a *try-catch statement*
- A *try block* is followed by one or more *catch* clauses
- Each catch clause has an associated exception type and is called an *exception handler*
- When an exception occurs within the try block, processing immediately jumps to the first catch clause that matches the exception type

Example 2

```
public class Example2 {
    public static void main(String[] args)
    {   Example2 ref=new Example2();
        try {
                ref.myMethod(5);
        }
        catch (AnException ex) {
            ex.printStackTrace();
        }
    }
    public void myMethod(int x) throws AnException
    {
        if (x>0) throw new AnException("Exception in
  method");
    }
}
```

The finally Clause

- A try statement can have an optional `finally` clause, which is always executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block finish
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause finish

Example 3

```
public class Example2 {
    public static void main(String[] args)
    {   Example2 ref=new Example2();
        try {
                  ref.myMethod(5);
        }
        catch (AnException ex) {
            ex.printStackTrace();
        }
        finally{
         // do something after an exception occurs or not
        }
    }
    public void myMethod(int x) throws AnException
    {
        if (x>0) throw new AnException("Exception in
  method");
    }
}
```

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the `main` method

Example 4

Method Call Propagation

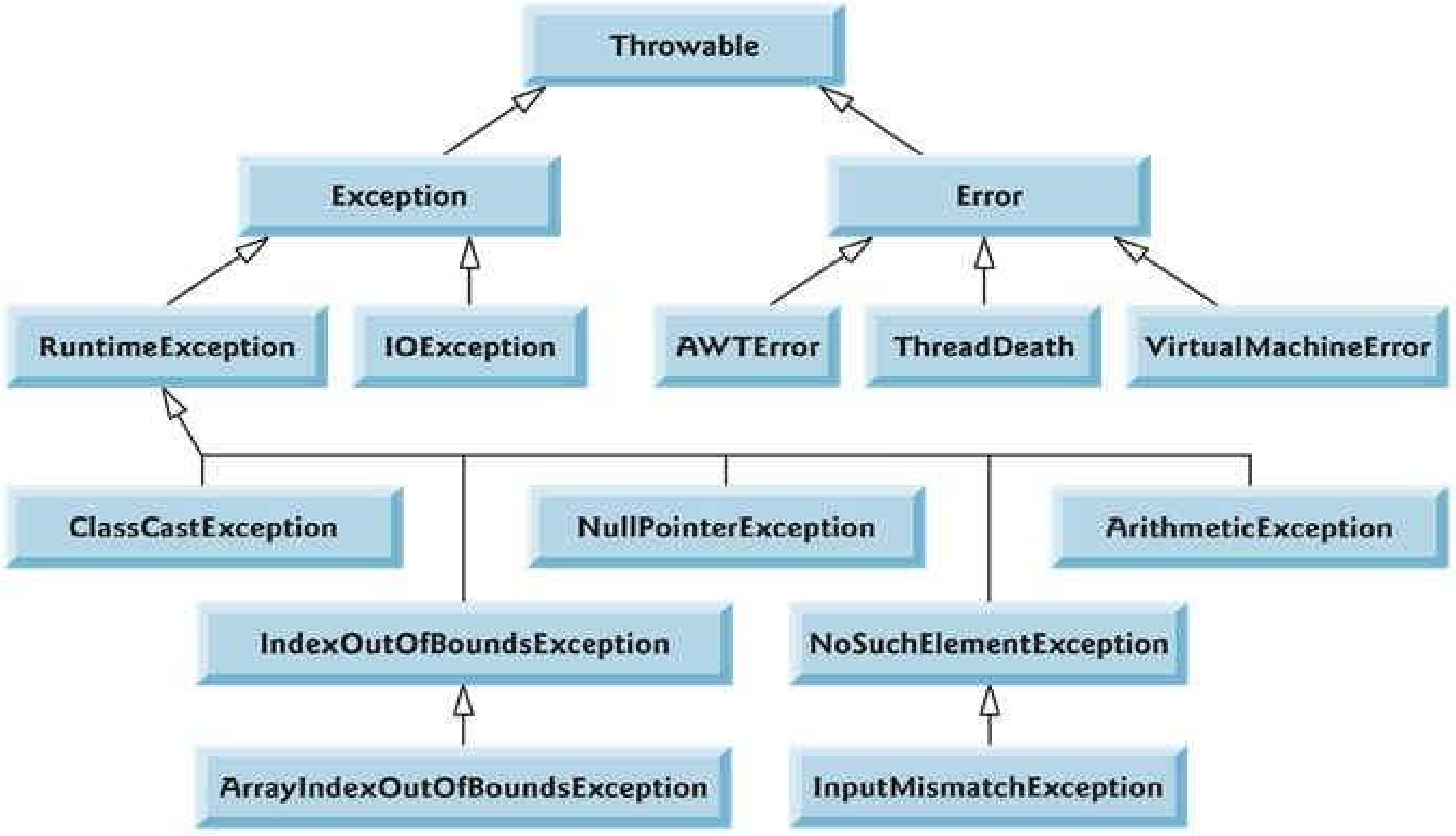
```
public class Example3
   public static void main(String[] args)
   {  Example3 ref=new Example3();
        try {
               ref.myMethod1();
        }catch (Exception ex) {
              ex.printStackTrace();
        }
   }
     public void myMethod1() throws Exception
     {
              myMethod(5);
     }
     public void myMethod(int x) throws Exception
     {
       if (x>0) throw new Exception("Exception in
                                          method");
     } }
```

Exception Propagation

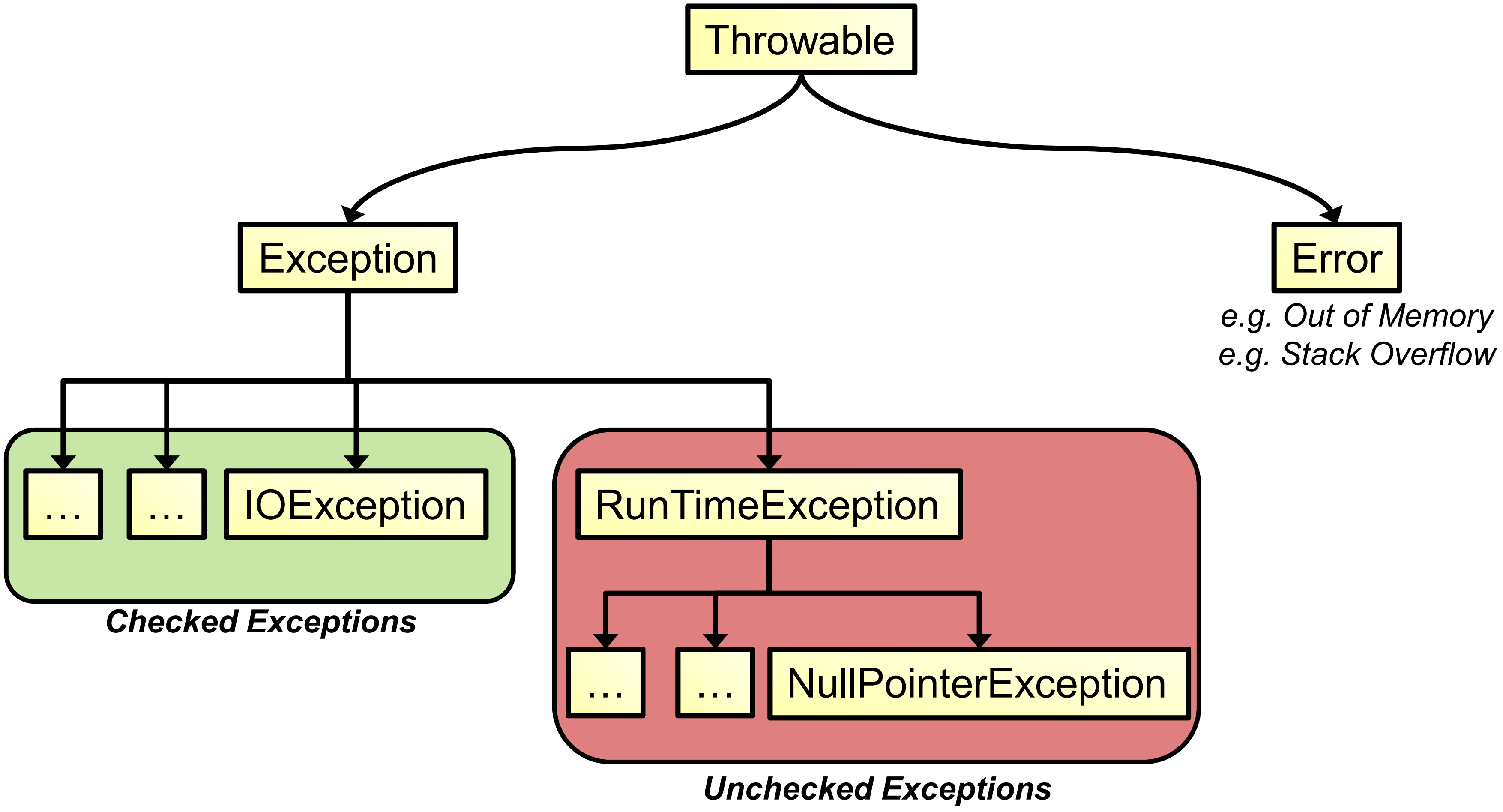
The Exception Class Hierarchy

- Exception classes in the Java API are related by inheritance, forming an exception class hierarchy
- All error and exception classes are descendents of the `Throwable` class
- A programmer can define an exception by extending the `Exception` class or one of its descendants
- The parent class used depends on how the new exception will be used

The Exception Class Hierarchy



The Exception Class Hierarchy



Checked Exceptions

- An exception is either *checked* or *unchecked*
- Are checked by the compiler
- A *checked exception* must either be caught or must be listed in the *throws clause* of any method that may throw or propagate it
- A throws clause is appended to the method header
- The compiler will issue an error if a checked exception is not caught or listed in a throws clause

Unchecked Exceptions

- An unchecked exception does not require explicit handling
- Are not checked by the compiler
- Are `RuntimeException` objects or descendants
- Errors
  - are similar to `RuntimeException` objects because
    - Errors should not be caught
    - Errors do not require a throws clause

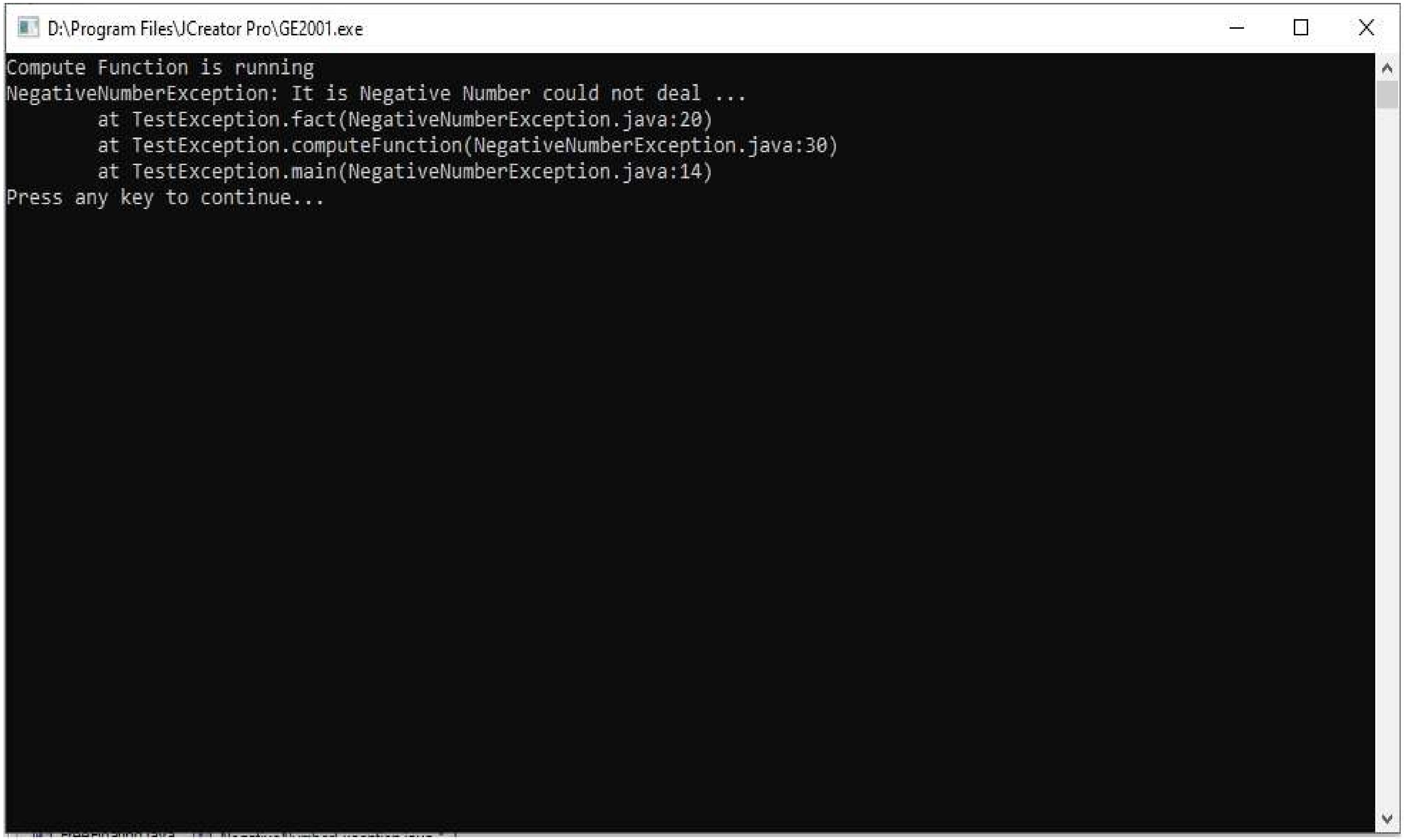
Example 5

```
public class NegativeNumberException extends Exception
{
        public NegativeNumberException()
        {
             super("It is Negative Number could not deal ...");
        }
}
class TestException
{
        public static void main(String []args)
        {
                        TestException t = new TestException();
                        t.computeFunction();
        }
```

Example 5

```
int fact(int n) throws NegativeNumberException
{       if(n<0)
                throw new NegativeNumberException();
        if(n==1) return 1;
        return n*fact(n-1);
}
public void computeFunction()
{
        System.out.println("Compute Function is running");
        try{
        System.out.println("Fact (-5) = " + fact(-5));
        }
        catch(NegativeNumberException e)
        {       e.printStackTrace();
        }
}
}
```

Example 5



```
Compute Function is running
NegativeNumberException: It is Negative Number could not deal ...
        at TestException.fact(NegativeNumberException.java:20)
        at TestException.computeFunction(NegativeNumberException.java:30)
        at TestException.main(NegativeNumberException.java:14)
Press any key to continue...
```

Considerations Regarding Exceptions

- If several method calls throw different exceptions,
  - then you can do either of the following:

1. Write separate `try-catch` blocks for each method.
2. Put them all inside the same `try` block and then write multiple `catch` blocks for it
   (one `catch` for each exception type).
3. Put them all inside the same `try` block and then just `catch` the parent of all exceptions: `Exception`.

Considerations Regarding Exceptions

- If more than one `catch` block are written after each other,
  - then you must take care not to handle a parent exception before a child exception
  - (i.e. a parent should not mask over a child).
  - Anyway, the compiler will give an error if you attempt to do so.

Example 6

```
try { ...... }
catch (FileNotFoundException e) {
  System.out.println("FileNotFoundException: ");
}
catch (IOException e) {
  System.out.println("Caught IOException: ");
}
```

In Java 7:

```
  try { ...... }
  catch (FileNotFoundException | IOException e) {
    System.out.println("....................");
  }
```

Considerations Regarding Exceptions cont'd

- An exception is considered to be one of the parts of a method's signature. So, when you override a method that throws a certain exception, you have to take the following into consideration:
  - You may throw the same exception.
  - You may throw a subclass of the exception
  - You may decide not to throw an exception at all
  - You **CAN'T** throw any different exceptions other than the exception(s) declared in the method that you are attempting to override
- A try block may be followed directly by a finally block.

The try-with-resources "Java 7"

- The try-with-resources statement:
  - is a try statement that declares one or more resources.
    - ***A resource:***
      - is an object that must be closed after the program is finished with it.
  - The try-with-resources statement ensures that each resource is closed at the end of the statement.
  - Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The try-with-resources example "Java 7"

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class Example2 {
   public static void main(String[] args)
   {
        try (BufferedReader br = new BufferedReader(new
                                 FileReader("C:\\testing.txt")))
        {
                String line;
                while ((line = br.readLine()) != null)
                {
                        System.out.println(line);
                }
        }
        catch (IOException e){
                e.printStackTrace();
        }
   }
}
```

Lab Exercise

1. Image on Applet

- Create an applet that loads and displays an image on it (.jpg or .gif).

- Scale the image to fit inside the applet if it is bigger than the applet's boundaries.

Hint: put the image file inside the same folder of the applet's .class file.

2. List of Fonts on Applet

- Create an applet that displays the list of available fonts in the underlying platform.

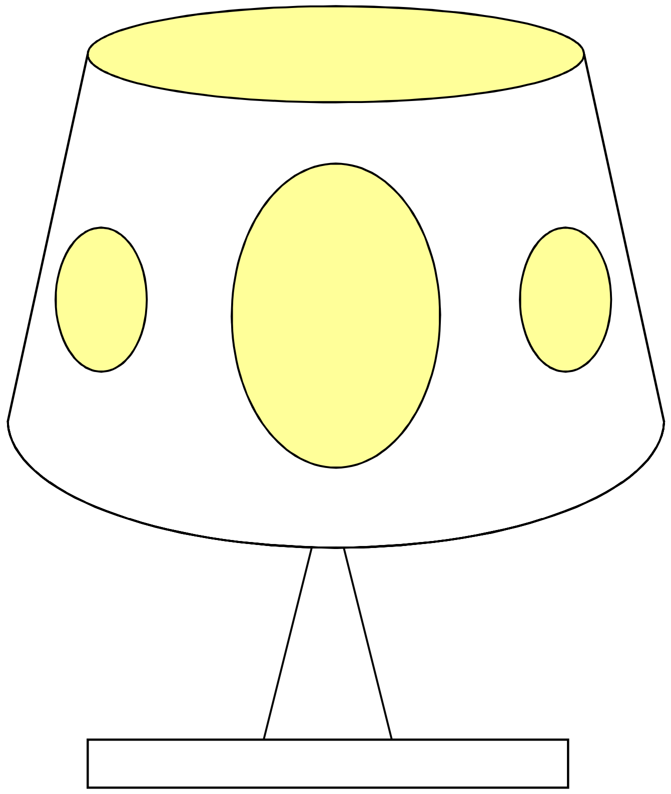
- Each font should be written in its own font.

- If you encounter any deprecated method|(s), follow the compiler instructions to re-compile and detect which method is deprecated. Afterwards, use the help (documentation) to see the proper replacement for the deprecated method(s).

Hint: You will need to set the height of the applet (in the HTML file) to a very large number (e.g. 2000!) in order to be able to view the whole list. Moreover, you'll need to browse the page using your web browser (e.g. Internet Explorer) in order to make use of the scroll bar.

3. Drawing a Lamp on Applet

- Create an applet that makes use of the Graphics class drawing methods.
- You may draw the following lamp:



Lesson 8

Interfaces

Interfaces

- In OOP, it is sometimes helpful to define what a class must do but not how it will do it.
- An abstract method defines the signature for a method but provides no implementation.
- A subclass must provide its own implementation of each abstract method defined by its superclass.
- Thus, an abstract method specifies the *interface* to the method but not the *implementation.*
- In Java, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces

- An ***interface*** is syntactically similar to an abstract class, in that you can specify one or more methods that have no body.
- Those methods must be implemented by a class in order for their actions to be defined.
- An *interface* specifies what must be done, but not how to do it.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement any number of interfaces.

Interfaces

Here is a simplified general form of a traditional interface:

```
access interface name
{
ret-type method-name1(param-list);
ret-type method-name2(param-list);
type var1 = value;
type var2 = value;
ret-type method-nameN(param-list);
type varN = value;
..
..

}
```

- Access specifier is either **public** or not used **(friendly)**
- methods are declared using only their return type and signature.
- They are, essentially, **abstract** methods and are implicitly **public**.
- Variables declared in an **interface** are not instance variables.
- Instead, they are implicitly **public**, **final**, and **static** and must be initialized.

Interfaces

Here is an example of an **interface** definition.

```
public interface Series
{
   int getNext();       // return next number in series
   void reset();           // restart
   void setStart(int x); // set starting value
}
```

Implementing Interfaces

The general form of a class that includes the **implements** clause looks like this:

```
access class classname extends superclass implements interface
{
            // class-body
}
```

Implementing Interfaces

```
// Implement Series.
class ByTwos implements Series
{       int start;
        int val;
        ByTwos(){
                start = 0;
                val = 0;
        }
        public int getNext() {
                val += 2;
                return val;
        }
        public void reset() {
                val = start;
        }
        public void setStart(int x) {
                start = x;
                val = x;
       }
}
```

- Class **ByTwos** implements the **Series** interface
- Notice that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier
- The next slide shows a class that demonstrates using the class ByTwos in a java applications

Implementing Interfaces

```
public class SeriesDemo {
  public static void main(String args[]) {
     ByTwos ob = new ByTwos();
     for (int i = 0; i < 5; i++) {
       System.out.println("Next value is " + ob.getNext());
     }
     System.out.println("\nResetting");
     ob.reset();
     for (int i = 0; i < 5; i++) {
       System.out.println("Next value is " + ob.getNext());
     }
     System.out.println("\nStarting at 100");
     ob.setStart(100);
     for (int i = 0; i < 5; i++) {
       System.out.println("Next value is " + ob.getNext());
    }
  }
}
```

Implementing Interfaces

```
// Implement Series.
class ByThrees implements Series
{       int start;
        int val;
        ByThrees() {
                start = 0;
                val = 0;
        }
        public int getNext() {
                val += 3;
                return val;
        }
        public void reset() {
                val = start;
        }
        public void setStart(int x){
                start = x;
                val = x;
        }
}
```

- Class **ByThrees** provides another implementation of the **Series** interface
- Notice that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier

Using interface reference

```
class SeriesDemo2
{
   public static void main(String args[])
   {
      ByTwos twoOb = new ByTwos();
      ByThrees threeOb = new ByThrees();
      Series ob;

      for(int i=0; i < 5; i++) {
       ob = twoOb;
       System.out.println("Next ByTwos value is " + ob.getNext());
       ob = threeOb;
     System.out.println("Next ByThrees value is " + ob.getNext());
      }
   }
}
```

General Consideration about interfaces

- **Variables** can be declared in an interface, but they are implicitly **public**, **static**, and **final**.
- To define a set of shared constants, create an **interface** that contains only these constants, without any methods.
- One **interface** can **inherit** another by use of the keyword **extends**. *The syntax is the same as for inheriting classes*.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Default Methods

Java SE 8 New Feature

- Prior to JDK 8, an interface *could not* define any implementation whatsoever.
- The release of JDK 8 changed this by adding a new capability to interface called the default method.
- A default method lets you define a *default implementation* for an interface method.
- You specify an interface default method by using the **default** keyword

Default Methods

Java SE 8
New Feature

```
public interface Series
{
// return first number in series
   int getNext()
   void reset();               // restart
   void setStart(int x);    // set starting value
   default int[] getNextArray(int n)
   {
        int[] vals = new int[n];
        for(int i=0; i < n; i++)
               vals[i] = getNext();
        return vals;
   }
}
```

Default Methods

Java SE 8
New Feature

- When you extend an interface that contains a default method, you can do the following:
  - Not mention the default method at all, which lets your extended interface inherit the default method.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {

}
```

Default Methods

Java SE 8 New Feature

- When you extend an interface that contains a default method, you can do the following:
  - Redefine the default method, which overrides it.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {
    default void method() { doAnother();  }
}
```

Default Methods

Java SE 8 New Feature

- When you extend an interface that contains a default method, you can do the following:
  - Re-declare the default method, which makes it abstract.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {
    abstract void method();
}
```

Use static Methods in an Interface

Java SE 8 New Feature

- JDK 8 added another new capability to interface: the ability to define one or more static methods.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- Thus, no implementation of the interface is necessary, and no instance of the interface is required in order to call a static method.
- The next slide shows an example

Use static Methods in an Interface

Java SE 8
New Feature

```
public interface MyIF {
// This is a "normal" interface method declaration.
//It does NOT define a default implementation.

   int getUserID();

// This is a default method. Notice that it provides a
//default implementation.
   default int getAdminID()
   {
       return 1;
   }
// This is a static interface method.
   static int getUniversalID()
   {
       return 0;
   }
}
```

Lesson 9

Multi-Threading

What is Thread?

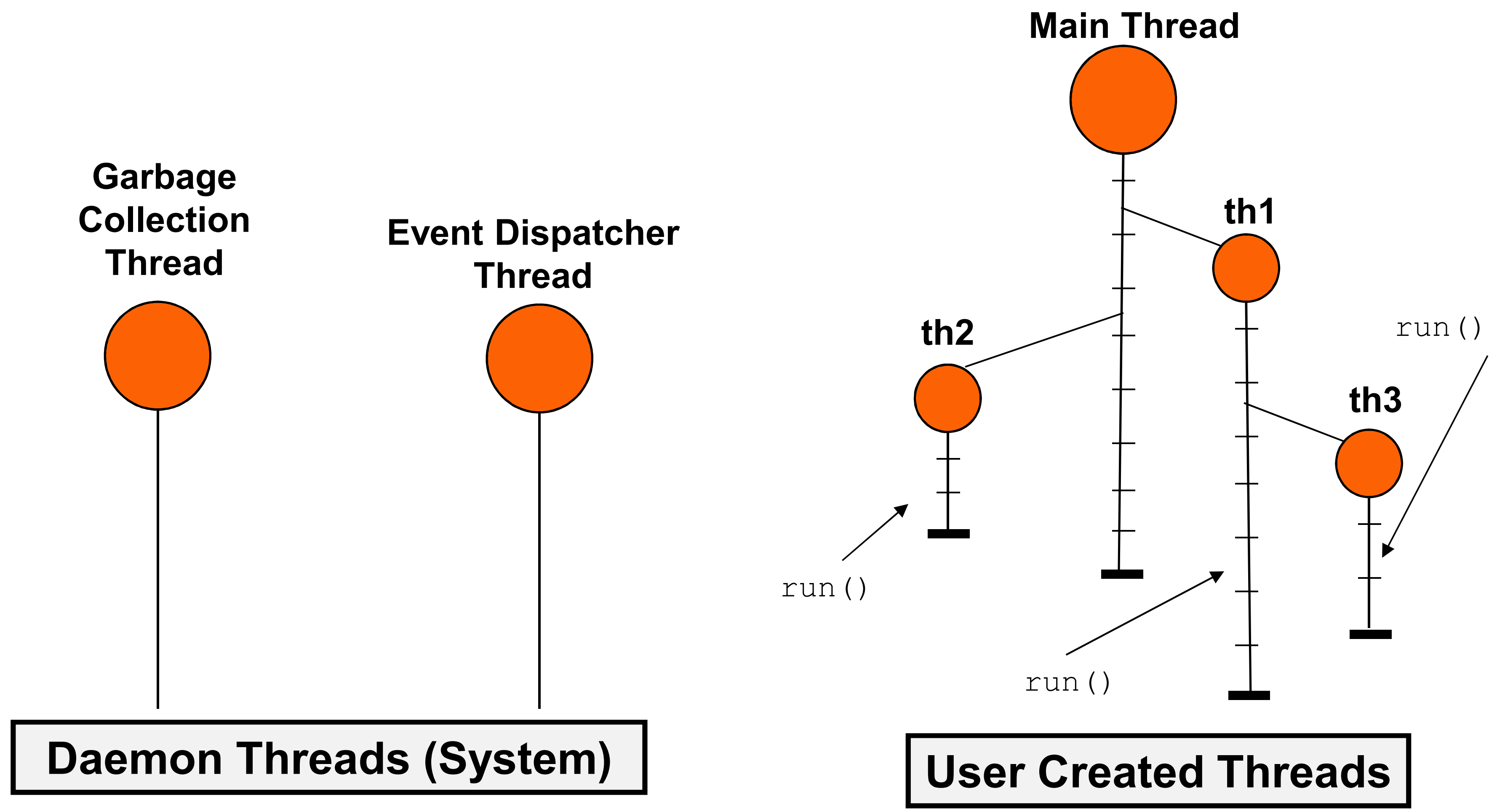
- A Thread is
  - A single sequential execution path in a program
  - Used when we need to execute two or more program segments concurrently (multitasking).
  - Used in many applications:
    - Games, animation, perform I/O
  - Every program has at least two threads.
  - Each thread has its own stack, priority & virtual set of registers.

What is Thread?

- Multiple threads do not means that they execute in parallel when you’re working in a single CPU.
  - Some kind of scheduling algorithm is used to manage the threads (e.g. Round Robin).
  - The scheduling algorithm is JVM specific (i.e. depending on the scheduling algorithm of the underlying operating system)

Threads

- several thread objects that are executing concurrently:



Threads cont'd

- Threads that are ready for execution are put in the ready queue.
  - Only one thread is executing at a time, while the others are waiting for their turn.
- The task that the thread carries out is written inside the `run()` method.

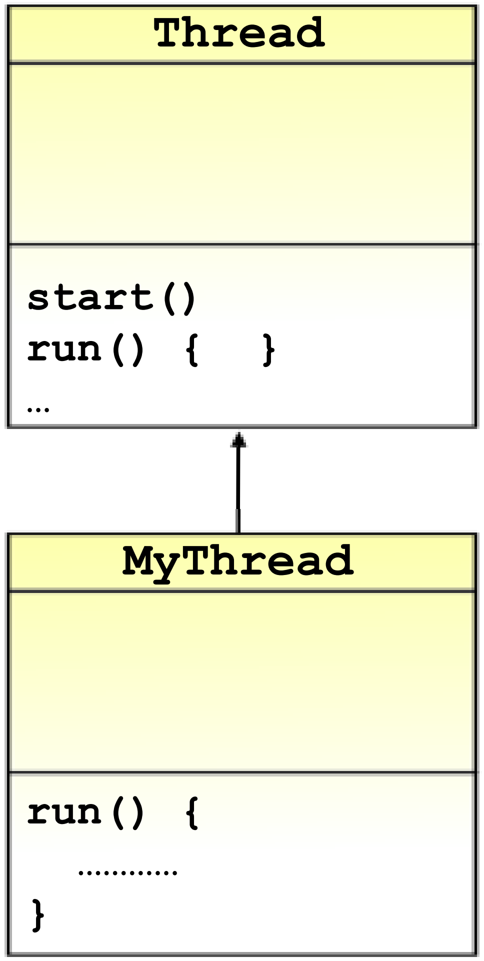
The Thread Class

- **Class Thread**
  - `start()`
  - `run()`
  - `sleep()`
  - `suspend()`*
  - `resume()`*
  - `stop()`*
- **Class Object**
  - `wait()`
  - `notify()`
  - `notifyAll()`

* *Deprecated Methods (may cause deadlocks in some situations)*

Working with Threads

- There are two ways to work with threads:
  - Extending Class **`Thread`**:
    1. Define a class that extends **`Thread`**.
    2. Override its **`run()`** method.
    3. In main or any other method:
       a. Create an object of the subclass.
       b. Call method **`start()`**.

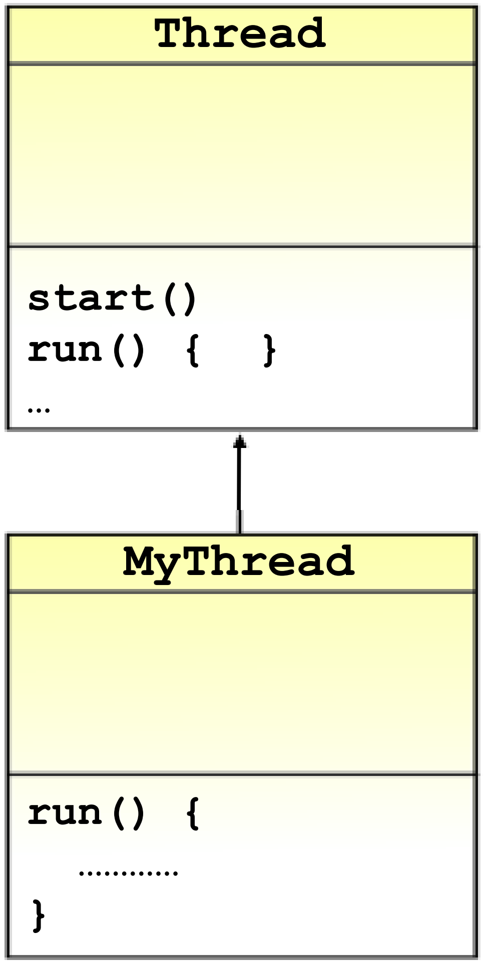


Working with Threads cont'd

```
public class MyThread extends Thread   1
{
   public void run()   2
   {
        … //write the job here
   }
}
```

- in **main()** or in the **init()** method of an applet or any method:

```
public void anyMethod()
{
   MyThread th = new MyThread();   3.a
   th.start();   3.b
}
```



Working with Threads

- There are two ways to work with threads:
  - Implementing Interface ***`Runnable`***:

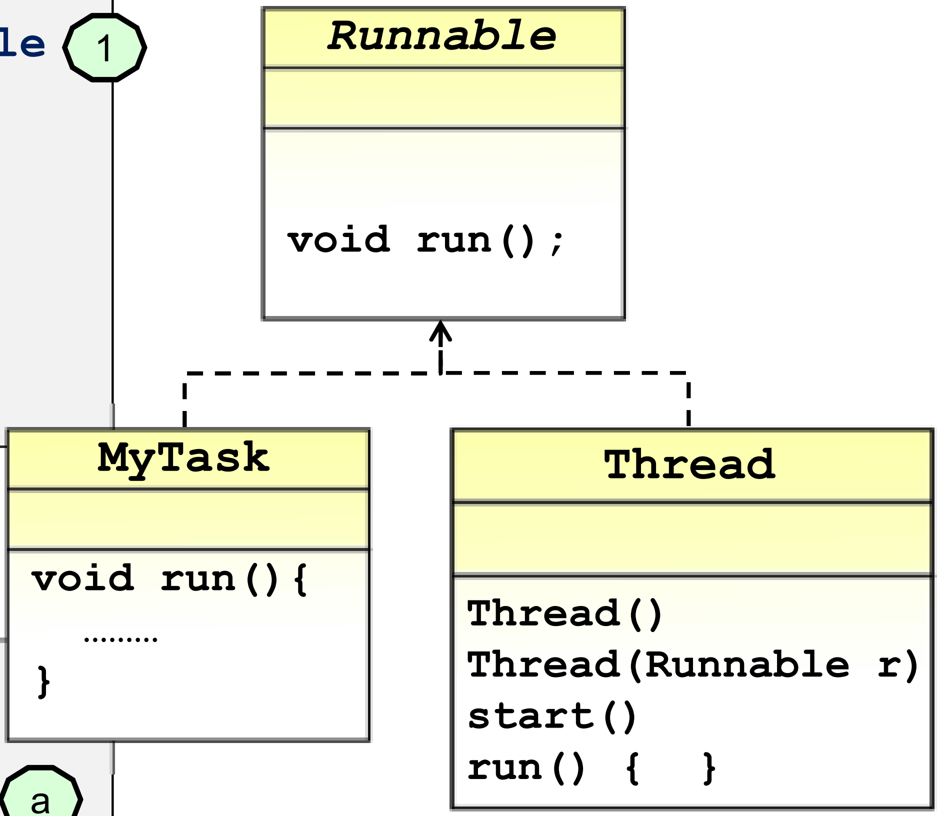
    1. Define a class that implements **`Runnable`**.
    2. Override its **`run()`** method .
    3. In main or any other method:
       a. Create an object of your class.
       b. Create an object of class **`Thread`** by passing your object to the constructor that requires a parameter of type **`Runnable`**.
       c. Call method **`start()`** on the **`Thread`** object.

Working with Threads cont'd

```
class MyTask implements Runnable   1
{
   public void run()   2
   {
        … //write the job here
   }
}
```

- in **main()** or in the **init()** method or any method:

3
```
public void anyMethod()
{
   MyTask task = new MyTask();   a
   Thread th = new Thread(task);   b
   th.start();   c
}
```



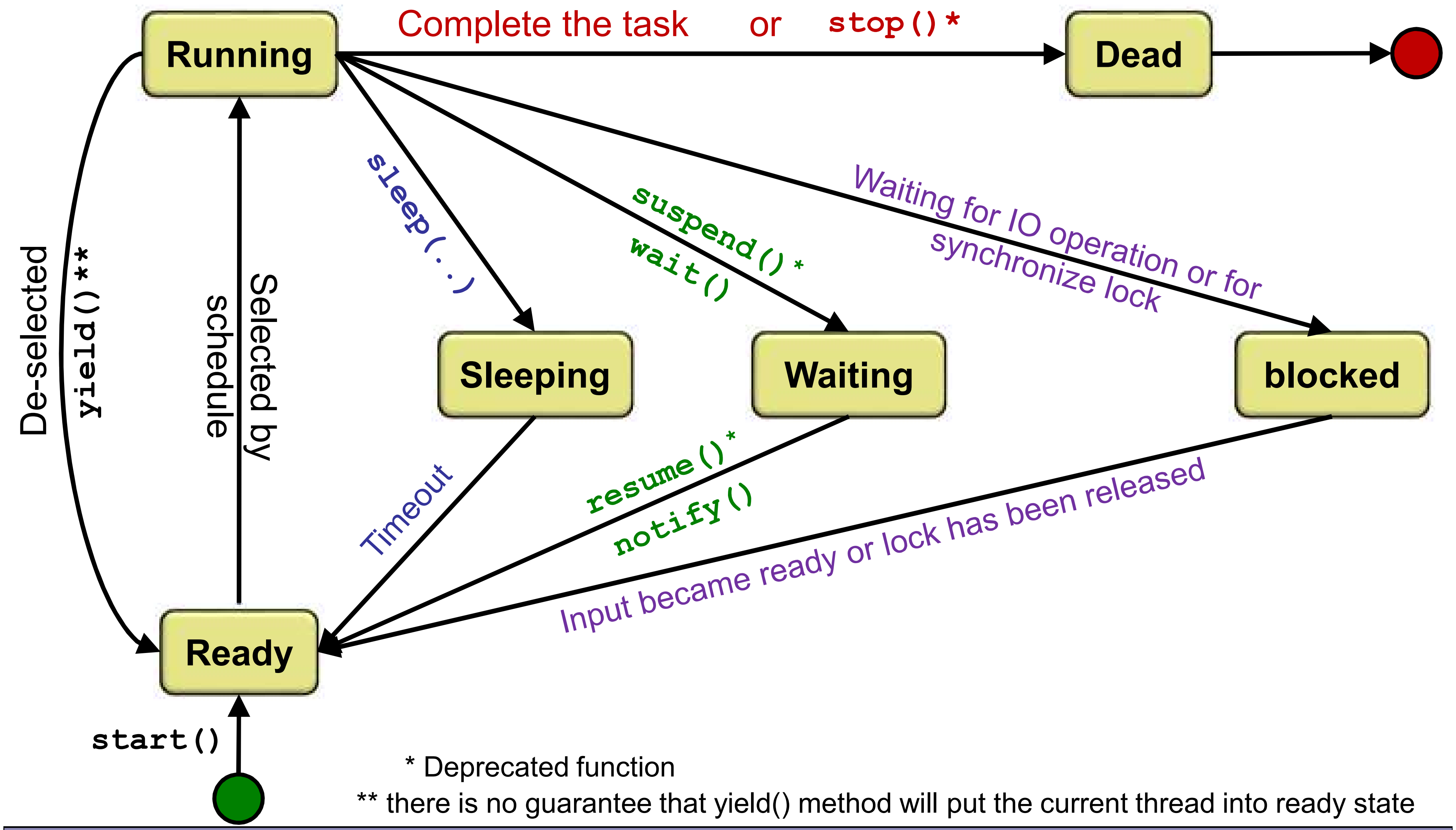
Extending Thread VS. Implementing Runnable

- Choosing between these two is a matter of taste.
- Implementing the Runnable interface:
  - May take more work since we still:
    - Declare a Thread object
    - Call the Thread methods on this object
  - Your class can still extend other class
- Extending the Thread class
  - Easier to implement
  - Your class can no longer extend any other class

Example: DateTimeApplet

```
public class DateTimeApp extends Applet implements Runnable{
   Thread th;  Date d;
public void init(){
     th = new Thread(this);
     th.start();
d = new Date();
   }
   public void paint(Graphics g){
        g.drawString(d.toString(), 50, 100);
   }
public void run(){
     while(true){
       d = new Date();
       repaint();
   Thread.sleep(1000); //you'll need to catch an exception here
     }
   }
}
```

Thread Life Cycle



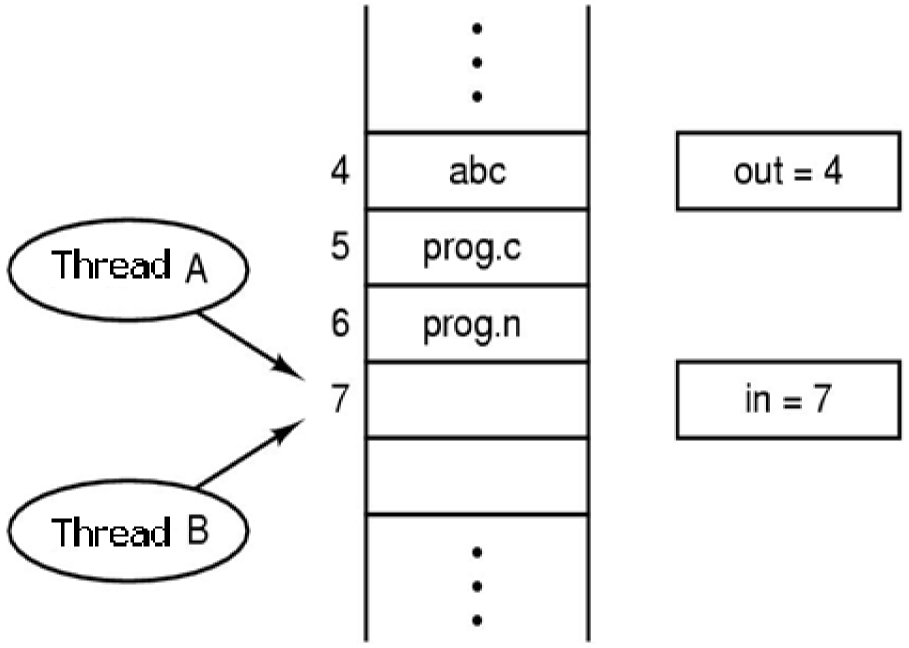
* Deprecated function

** there is no guarantee that yield() method will put the current thread into ready state

Synchronization

- Race conditions occur when
  - Multiple threads access the same object (shared resource)
- Example:

  Two threads want to access the same file, one thread reading from the file while another thread writes to the file.



They can be avoided by synchronizing the threads which access the shared resource.

Unsynchronized Example

```
class TwoStrings {
    static void print(String str1, String str2) {
        System.out.print(str1);
        try { Thread.sleep(500); }
        catch (Exception e) { e.printStackTrace(); }
        System.out.println(str2);
    }
}
class PrintStringsThread implements Runnable {
    Thread thread;
    String s1, s2;
    PrintStringsThread(String str1, String st
        s1 = str1;
        s2 = str2;
        thread = new Thread(this);
        thread.start();
    }
    public void run() { TwoStrings.print(str1, str2); }
}
class Test {
    public static void main(String args[]) {
        new PrintStringsThread("Hello ", "there.");
        new PrintStringsThread("How are ", "you?");
        new PrintStringsThread("Thank you ", "very much!");
    }
}
```

- Sample output:

```
Hello How are Thank you there.
you?
very much!
```

Synchronized method

```
class TwoStrings {
    synchronized static void print(String str1, String str2) {
        System.out.print(str1);
        try { Thread.sleep(500); }
        catch (Exception e) { e.printStackTrace(); }
        System.out.println(str2);
    }
}
class PrintStringsThread implements Runnable {
    Thread thread;
    String s1, s2;
    PrintStringsThread(String str1, String str2) {
        s1 = str1;
        s2 = str2;
        thread = new Thread(this);
        thread.start();
    }
    public void run() { TwoStrings.print(str1, str2); }
}
class Test {
    public static void main(String args[]) {
        new PrintStringsThread("Hello ", "there.");
        new PrintStringsThread("How are ", "you?");
        new PrintStringsThread("Thank you ", "very much!");
    }
}
```

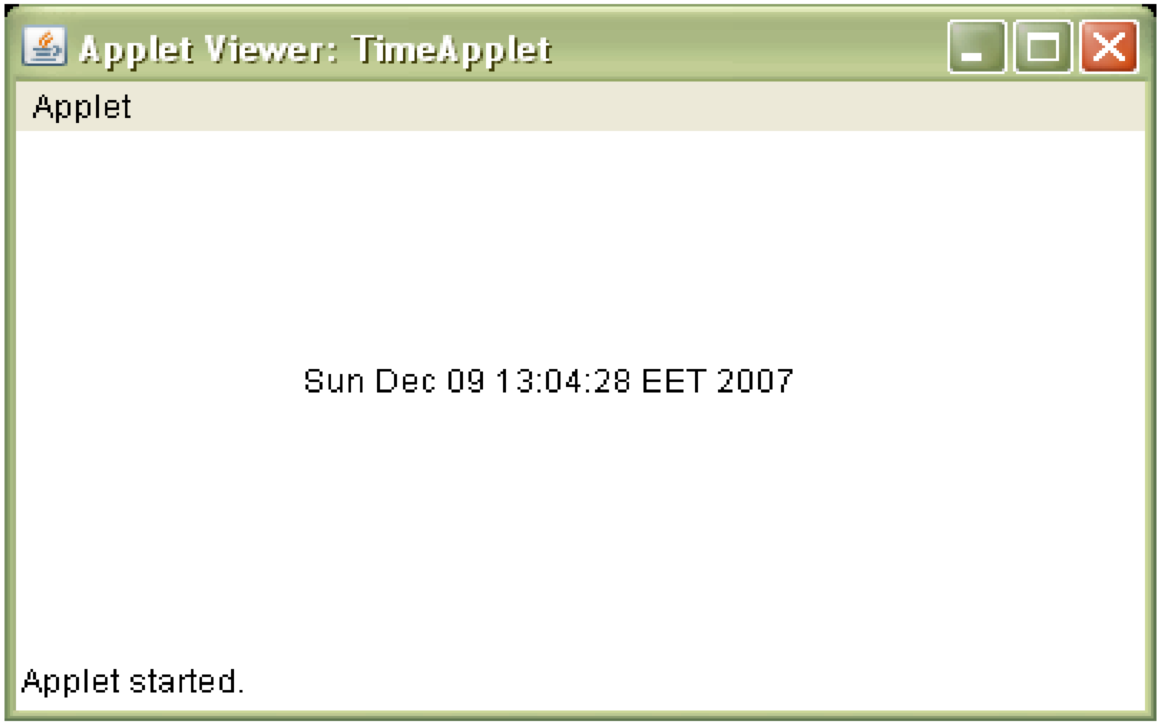
- Sample output:

```
Hello there.
How are you?
Thank you very much!
```

Lab Exercise

1. Date and Time Applet

- Create an applet that displays date and time on it.

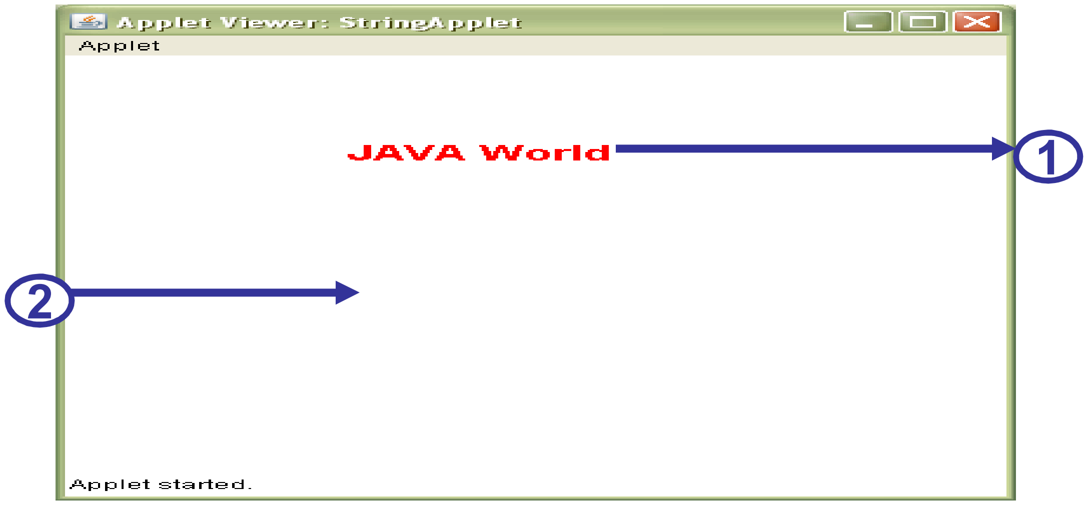


```
Sun Dec 09 13:04:28 EET 2007
```

Applet started.

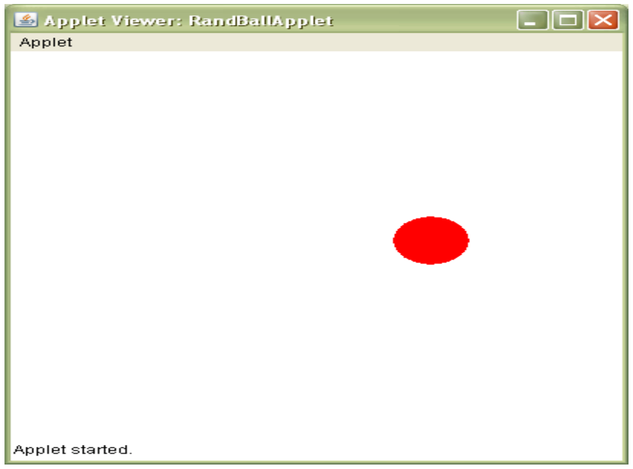
2. Text Banner Applet

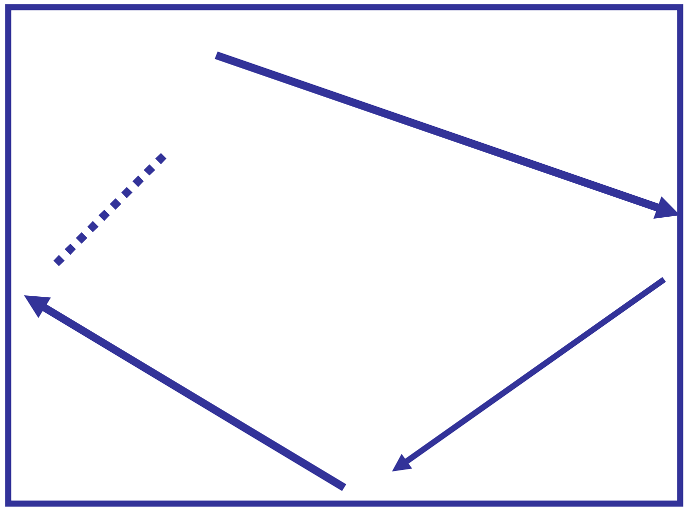
•Create an applet that displays marquee string on it.



3. Animation Ball Applet

•Create an applet that displays ball which moves randomly on this applet.





Lesson 10

Inner Classes

Inner Classes

- The Java programming language allows you to define a class within another class.
  - Such a class is called a ***nested class [Inner Class]***.

```
class OuterClass
 {
        ...
        class InnerClass
        {
               ...
        }
}
```

Why Use Inner Classes?

- There are several reasons for using inner classes:

  1. It is a way of logically grouping classes that are only used in one place.
     - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
     - Nesting such "helper classes" makes their package more streamlined.

Why Use Inner Classes?

- There are several reasons for using inner classes:

2. It increases encapsulation.
    - Consider two top-level classes, A and B, where B needs access to private members of A. By hiding class B within class A, A's members can be declared private and B can access them.
    - In addition, B itself can be hidden from the outside world.

```
class A
 {
          ...
          class B
          {
                  ...
          }
}
```

Why Use Inner Classes?

- There are several reasons for using inner classes:

3. Nested classes can lead to more readable and maintainable code.

    - Nesting small classes within top-level classes places the code closer to where it is used.

Types of Inner Classes

- There are broadly four types of inner classes:

  1. Normal Member Inner Class
  2. Static Member Inner Class
  3. Local Inner Class (inside method body)
  4. Local Anonymous Inner Class

1. Normal Member Inner Class

```
public class MyClass{
  private int x ;
  public void myMethod(){
      MyInnerClass m = new MyInnerClass();
      m.aMethod();
      ..
  }
  class MyInnerClass{
      public void aMethod(){
          //you can access private members of the outer class here
          x = 3 ;
      }
  }
}
```

1. Normal Member Inner Class

- In order to create an object of the inner class you need to use an object of the outer class.
- The following line of code could have been written inside the method of the enclosing class:

```
MyInnerClass m = this.new MyInnerClass();
```

- The following line of code is used to create an object of the inner class outside of the enclosing class:

```
OuterClass obj = new OuterClass() ;
OuterClass.MyInnerClass m = obj.new MyInnerClass();
```

1. Normal Member Inner Class

- An inner class can extend any class and/or implement any interface.
- An inner class can assume any accessibility level:
  - private, (friendly), protected, or public.
- An inner class can have an inner class inside it.
- When you compile the java file, two class files will be produced:
  - MyClass.class
  - MyClass$MyInnerClass.class
- The inner class has an implicit reference to the outer class.

1. Normal Member Inner Class

The inner class has an implicit reference to the outer class

```
public class MyClass{
   private int x ;
   public void myMethod(){
        MyInnerClass m = new MyInnerClass();
        m.aMethod();
   }
   class MyInnerClass{
        int x ;
        public void aMethod(){
             x = 10 ;  //x of the inner class
             MyClass.this.x = 25 ;  // x of the outer class
        }
   }
}
```

Example

```
public class DateTimeApp extends Applet{
   Thread th; Date d;
   public void init(){
     d  = new Date();
     th = new DateThread();
     th.start();
   }
   public void paint(Graphics g){
      g.drawString(d.toString(), 50, 100);
   }
   class DateThread extends Thread{
       public void run(){
              while(true){
               d = new Date();
              repaint();
           Thread.sleep(1000); // need to catch an exception here
              }
       }
  }}
```

Example

```
public class DataStructure {
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];
    public void printEven() {
        InnerEvenIterator iterator = this.new InnerEvenIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }
    private class InnerEvenIterator {
        private int next = 0;
        public boolean hasNext() {
            return (next <= SIZE - 1);
        }
        public int getNext() {
            int retValue = arrayOfInts[next];
            next += 2;
            return retValue;
        }
    }
    public static void main(String s[]) {
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

2. Static Inner Class

- You know, The normal inner class has implicitly a reference to the outer class that created it.
  - If you don't need a connection between them, then you can make the **inner class static**.
- **A static inner class** means:
  - You don't need an outer-class object in order to create an object of a static inner class.
  - You can't access an outer-class object from an object of a static inner class.

2. Static Inner Class

- Static Inner Class:
  - is among the static members of the outer class.
  - When you create an object of static inner class, you don't need to use an object of the outer class (remember: it's static!).
  - Since it is static, such inner class will only be able to access the static members of the outer class.

2. Static Inner Class (Example)

```
public class OuterClass{

  int x ;
  static int y;

  static class InnerClass{
     y = 10;          // OK
     x = 33;          // wrong
  }
}
```

```
InnerClass ic= OuterClass.new InnerClass();
```

3. Local Inner Class

```
public class MyClass {
  private int x ;
  public void myMethod(final String str, final int a){
      final int b;
      class MyLocalInnerClass{
          public void aMethod(){
        //you can access private members of the outer class
       //and you can access final local variables of the method
          }
      }
     MyLocalInnerClass myObj = new MyLocalInnerClass();
   }
}
```

3. Local Inner Class

- The object of the local inner class can only be created below the definition of the local inner class (within the same method).
- The local inner class can access the member variables of the outer class.
- It can also access the local variables of the enclosing method if they are declared final.

4. Anonymous Inner Class

```
public class MyClass extends Applet
{
   int x ;
   public void init()
   {
     Thread th = new Thread(new Runnable()
                              {
                                 public void run()
                                 {
                                     ..
                                 }
                              });
     th.start();
   }
}
```

4. Anonymous Inner Class

- The whole point of using an anonymous inner class is to implement an interface or extend a class and then override one or more methods.
- Of course, it does not make sense to define new methods in anonymous inner class; how will you invoke it?
- When you compile the java file, two class files will be produced:
  - MyClass.class
  - MyClass$1.class

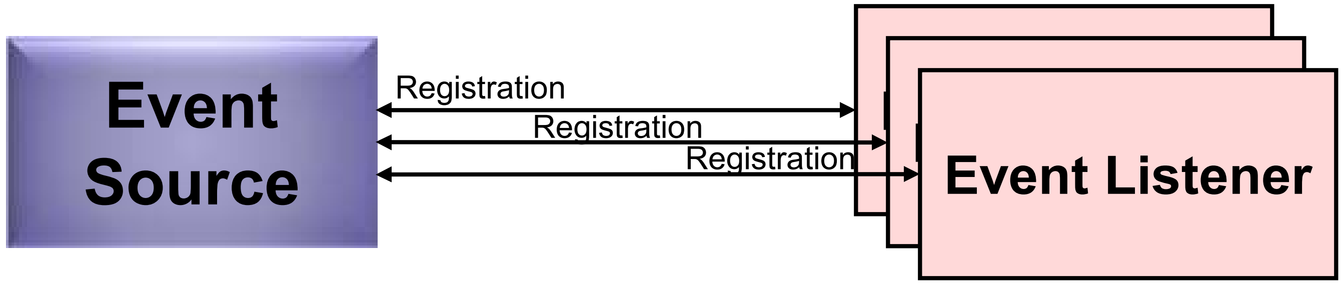
Lesson 11

Event Handling

Event Handling

- Event Delegation Model was introduced to Java since JDK 1.1
- This model realizes the event handling process as two roles:
  - Event Source
  - Event Listener.
- The Source:
  - is the object that fired the event,
- The Listener:
  - is the object that has the code to execute when notified that the event has been fired.

Event Handling



- An Event Source may have one or more Event Listeners.
- The advantage of this model is:
  - The Event Object is only forwarded to the listeners that have registered with the source.

Event Handling

- When working with GUI, the source is usually one of the GUI Components (e.g. Button, Checkbox, …etc).
- The following piece of code is a simplified example of the process:

```
Button b = new Button(“Ok”); //Constructing a Component (Source)
MyListener myL = new MyListener(); //Constructing a Listener
b.addActionListener(myL); //Registering Listener with Source
```

Event Handling

- Let's look at the signature of the registration method:

```
void addActionListener(ActionListener l)
```

- In order for a listener to register with a source for a certain event,
  - it has to implement the proper interface that corresponds to the designated event, which will enforce a certain method to exist in the listener.

Event Handling Example

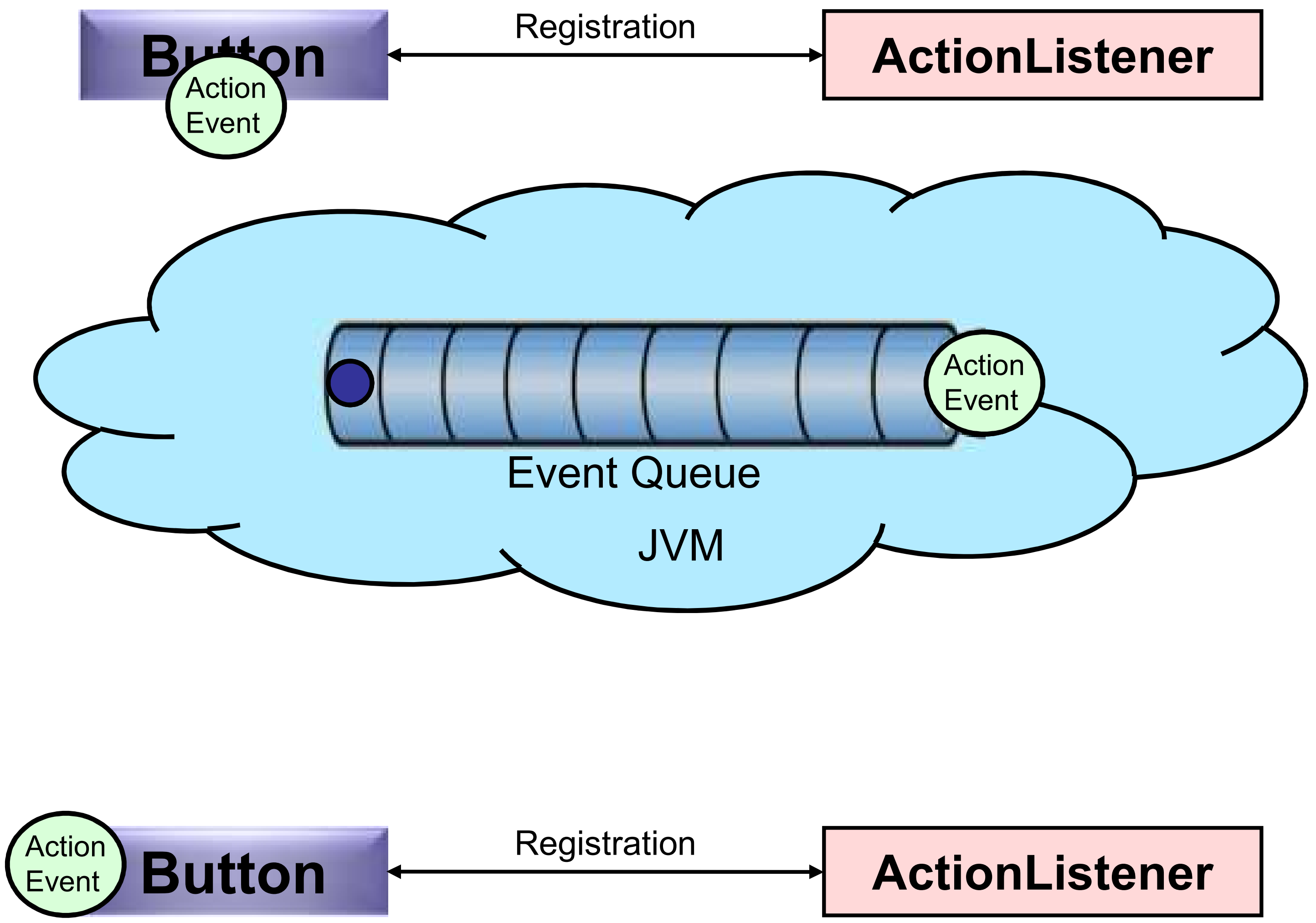
1. Write the listener code:

```
class MyListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
       // handle the event here
       // (i.e. what you want to do
       // when the Ok button is clicked)
   }
}
```

2. Create the source, and register the listener with it:

```
public class MyApplet extends Applet{
       public void init(){
           Button b = new Button(“Ok”);
           MyListener myL = new MyListener();
           b.addActionListener(myL);
       }
}
```

Event Dispatching Thread



Event Dispatching Thread

- When examining the previous button example:
  - The button (**source**) is created.
  - The listener is registered with the button,
  - The user clicks on the Ok button.
    - An ActionEvent object is created and placed on the **event queue**.
  - The **Event Dispatching Thread** processes the events in the queue.
    - The Event Dispatching Thread checks with the button to see if any listeners have registered themselves.
  - The Event Dispatcher then invokes the actionPerformed(..) method, and passes the ActionEvent object itself to the method.

Event Handling Example

```
public class ButtonApplet extends Applet{
   int x = 0;
   Button b;
   public void init(){
        b = new Button("Click Me");
        b.addActionListener(new MyButtonListener());
        add(b);
   }
   public void paint(Graphics g){
        g.drawString("Click Count is:" + x, 50, 200);
   }
   class MyButtonListener implements ActionListener{
        public void actionPerformed(ActionEvent ev){
             x++ ;
             repaint() ;
        }
   }
}
```

Event Handling Example

```
public class ButtonApplet extends Applet{
   int x = 0;
   Button b;
   public void init(){
        b = new Button(“Click Me”);
        b.addActionListener(
        new ActionListener(){
                public void actionPerformed(ActionEvent ev){
                                x++ ;
                                repaint() ;
                        }
                });
        add(b);
   }
   public void paint(Graphics g){
        g.drawString(“Click Count is:“ + x, 50, 200);
   }
}
```

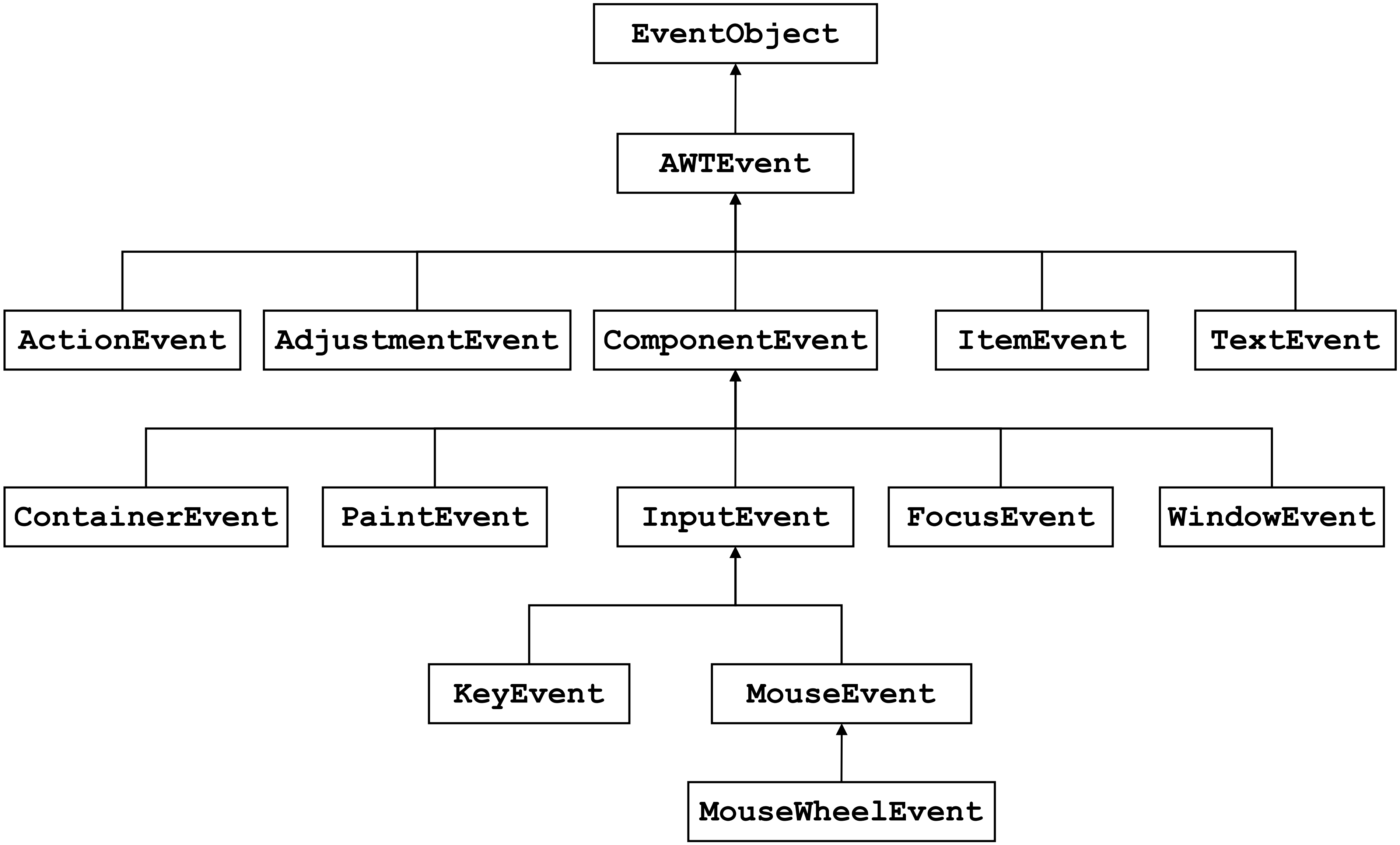
Event Handling Example

```
public class ButtonApplet extends Applet{
   int x = 0;
   Button b;
   public void init(){
        b = new Button("Click Me");
        b.addActionListener((ActionEvent ev)->{
                                        x++ ;
                                        repaint() ;
                                  }
                    });
        add(b);
   }
   public void paint(Graphics g){
        g.drawString("Click Count is:" + x, 50, 200);
   }
}
```

Event Handling Example

```
public class ButtonApplet extends Applet{
   int x = 0;
   Button b;
   public void init(){
        b = new Button(“Click Me”);
        b.addActionListener((ev)->{
                                        x++ ;
                                        repaint() ;
                                 }
                    });
        add(b);
   }
   public void paint(Graphics g){
        g.drawString(“Click Count is:“ + x, 50, 200);
   }
}
```

Event Class Hierarchy



Events Classes and Listener Interfaces

Event	Listener Interface(s)	Method(s)
ActionEvent	ActionListener	actionPerformed (ActionEvent e)
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent e)
ComponentEvent	ComponentListener	componentHidden (ComponentEvent e) componentShown (ComponentEvent e) componentMoved (ComponentEvent e) componentResized (ComponentEvent e)
ItemEvent	ItemListener	itemStateChanged (ItemEvent e)
TextEvent	TextListener	textValueChanged (TextEvent e)
ContainerEvent	ContainerListener	componentAdded (ContainerEvent e) componentRemoved (ContainerEvent e)

Events Classes and Listener Interfaces

Event	Listener Interface(s)	Method(s)
FocusEvent	FocusListener	focusGained (FocusEvent e) focusLost (FocusEvent e)
WindowEvent	WindowListener	windowClosed (WindowEvent e) windowClosing (WindowEvent e) windowOpened (WindowEvent e) windowActivated (WindowEvent e) windowDeactivated (WindowEvent e) windowIconified (WindowEvent e) windowDeiconfied (WindowEvent e)

Events Classes and Listener Interfaces

Event	Listener Interface(s)	Method(s)
KeyEvent	KeyListener	keyPressed (KeyEvent e) keyReleased (KeyEvent e) keyTyped (KeyEvent e)
MouseEvent	MouseListener	mousePressed (MouseEvent e) mouseReleased (MouseEvent e) mouseClicked (MouseEvent e) mouseEntered (MouseEvent e) mouseExited (MouseEvent e)
	MouseMotionListener	mouseMoved (MouseEvent e) mouseDragged (MouseEvent e)
MouseWheel Event	MouseWheelListener	mouseWheelMoved (MouseWheelEvent e)

Adapters

- When working with listener interfaces that have more than one method,
  - it is a tedious task to override all the methods of the interface when we only need to implement one of them.
- Therefore, for each listener interface with multiple methods,
  - there is a special **Adapter** class that has implemented the interface and overridden all the methods with empty bodies.

Adapters

- You then only need to extend the corresponding Adapter class instead of implementing the interface, then overriding the required methods only.
- For example, the **`WindowListener`** interface has a corresponding **`WindowAdapter`** class.

Another Event Handling Example

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

public class SampleUI extends JFrame {
   public SampleUI(){
       this.setLayout(new FlowLayout());
       JTextArea ta = new JTextArea(20,50);
       JScrollPane scroll = new JScrollPane(ta);
       JTextField tf = new JTextField(40);
       JButton okButton = new JButton("Send");

       okButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent ae){
                    ta.append(tf.getText()+"\n");
                    tf.setText(“”);
            }
        });
```

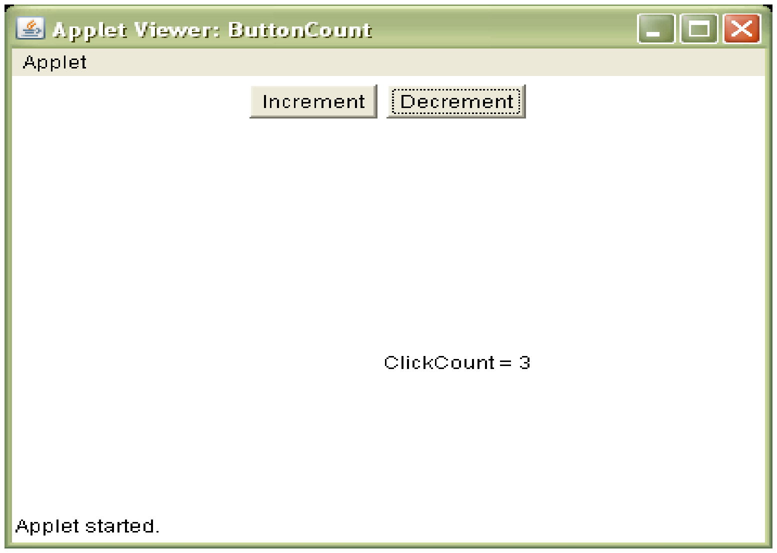
Another Event Handling Example

```
        add(scroll);
        add(tf);
        add(okButton);
    }
    public static void main(String args[])
    {
        SampleUI ui=new SampleUI();
        ui.setSize(600, 400);
        ui.setResizable(false);
        ui.setVisible(true);
     }
}
```

Lab Exercise

1. Button Count Applet

- Create an applet that has two buttons one to increment the counter value and one to decrement this value.

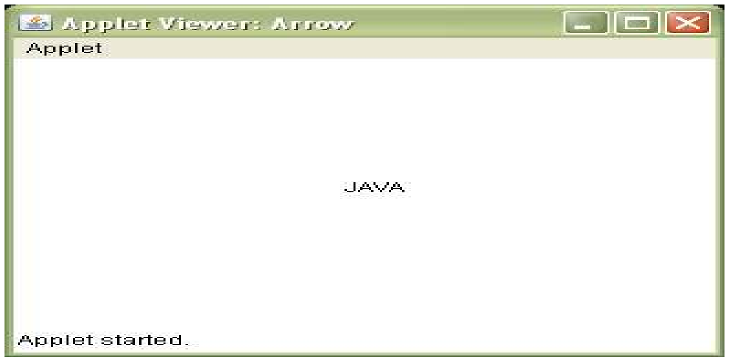


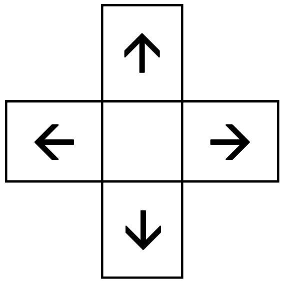
ClickCount = 3

Applet started.

2. Moving Text Using Keyboard

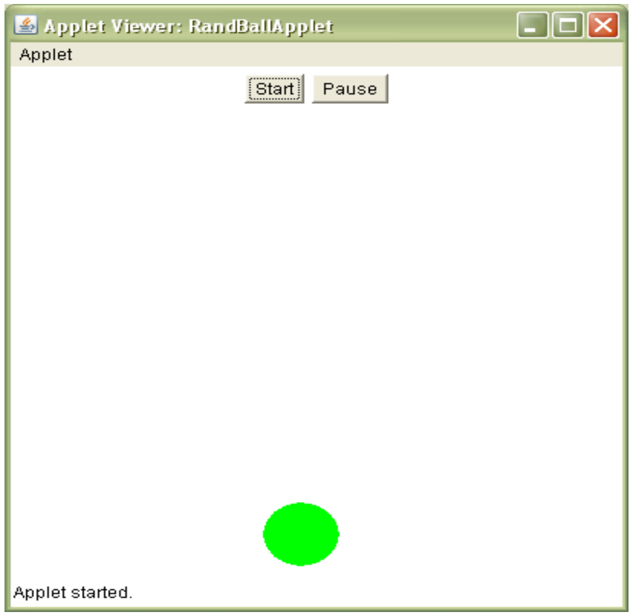
- Create an applet that displays string which user can move it using arrow keys.





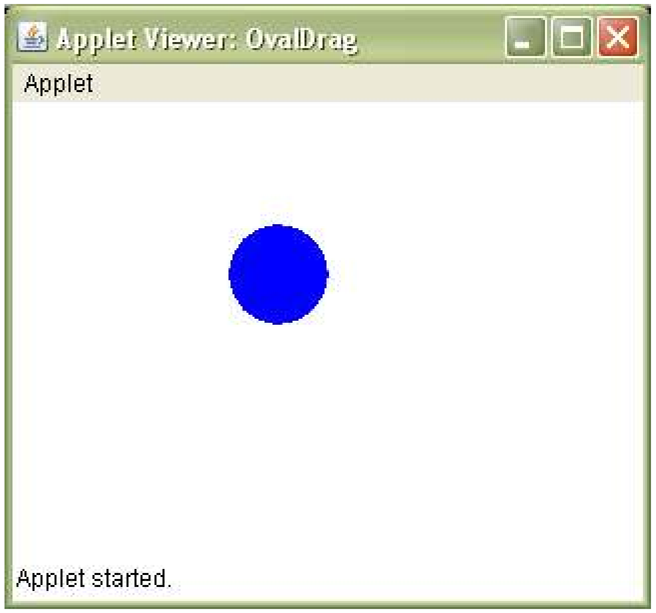
3. Play and Pause Animation

- Create an applet that has two buttons one to let ball star moving randomly and one to pause this ball moving.



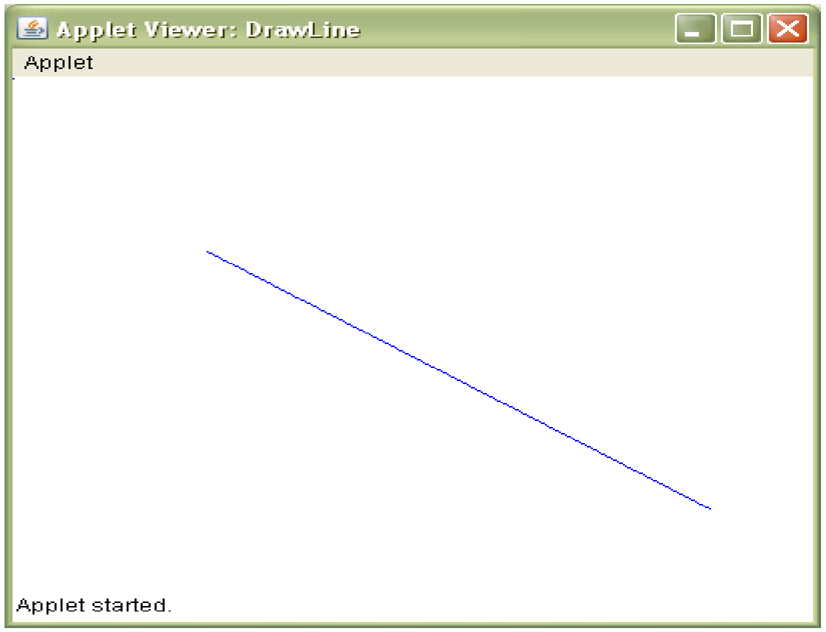
4. Drag Ball Applet

- Create an applet that draws an oval which the user can drag around the applet.



5. Draw Single Line Applet

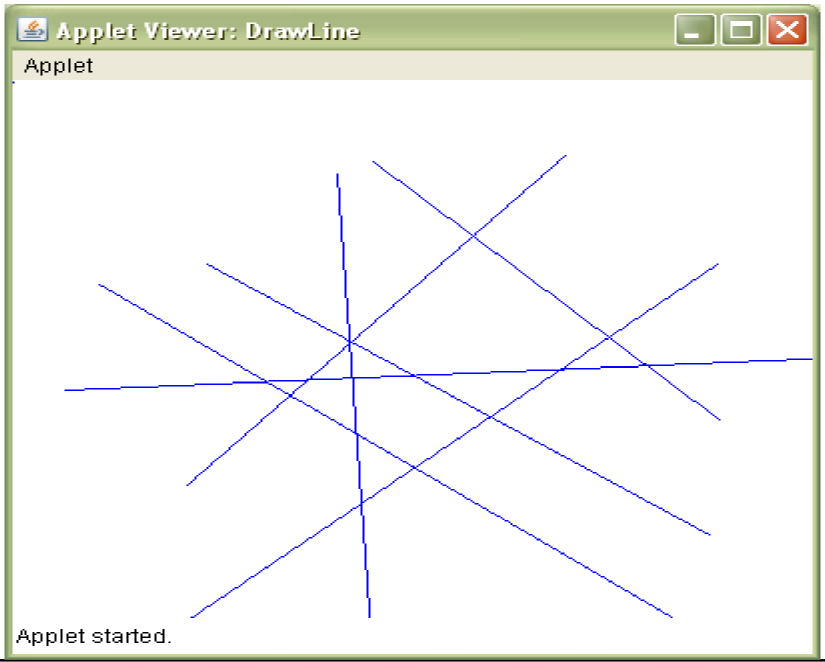
- Create an applet that allows the user to draw one line by dragging the mouse on the applet.



6. Draw Multiple Lines Applet

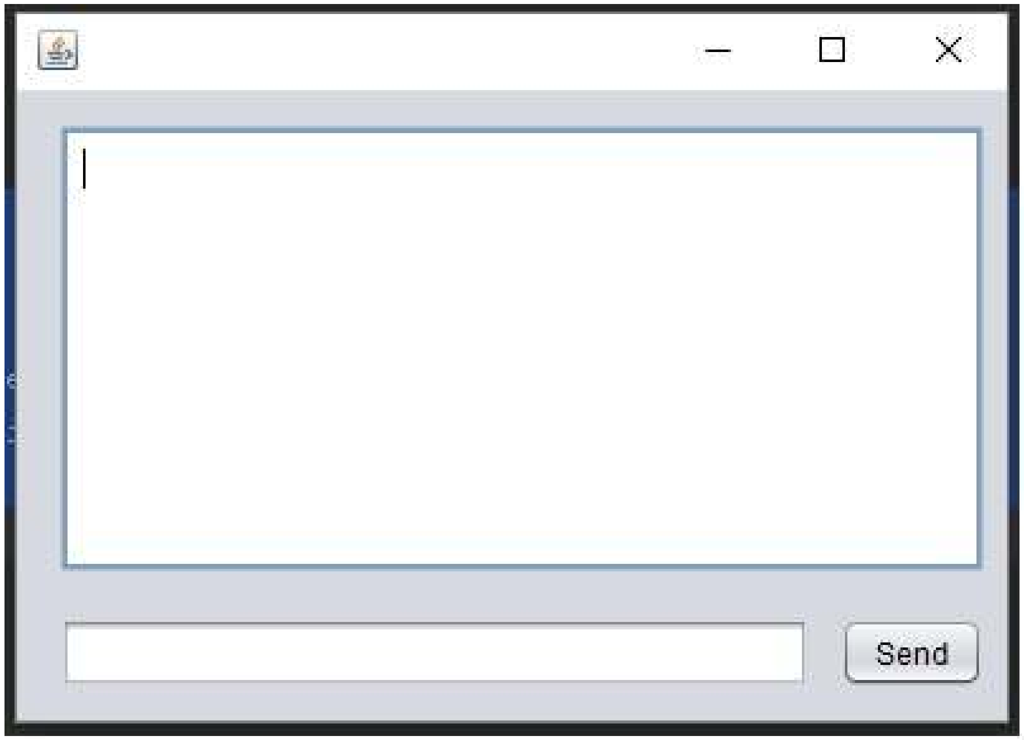
- Modify the previous exercise to allow the user to draw multiple lines on the applet.

- Store the drawn lines in an array to be able to redraw them again when the applet is repainted.



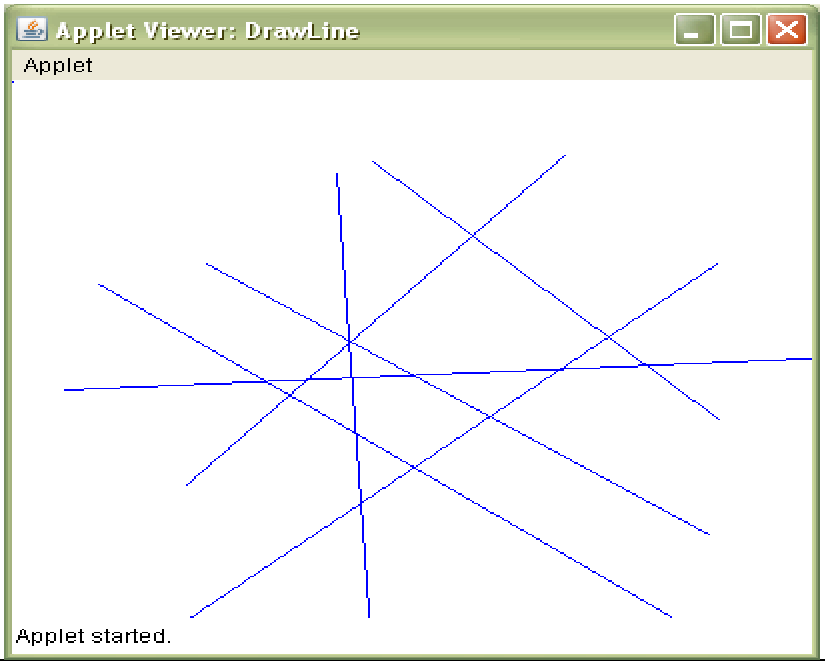
7. Create A GUI Desktop Application

- Create a GUI Desktop Application which you can use as a simple chatting program interface .



7. Draw Multiple Lines – using Vector

- In the pervious exercise the user can only draw a certain number of lines because of the fixed array size.
  - Modify the pervious exercise, by storing the lines in a Vector, to allow an unlimited number.


Applet Viewer: DrawLine
Applet
Applet started.