



# Java™ Education & Technology Services

## Java Programming



# Lesson 1

## Introduction To Java

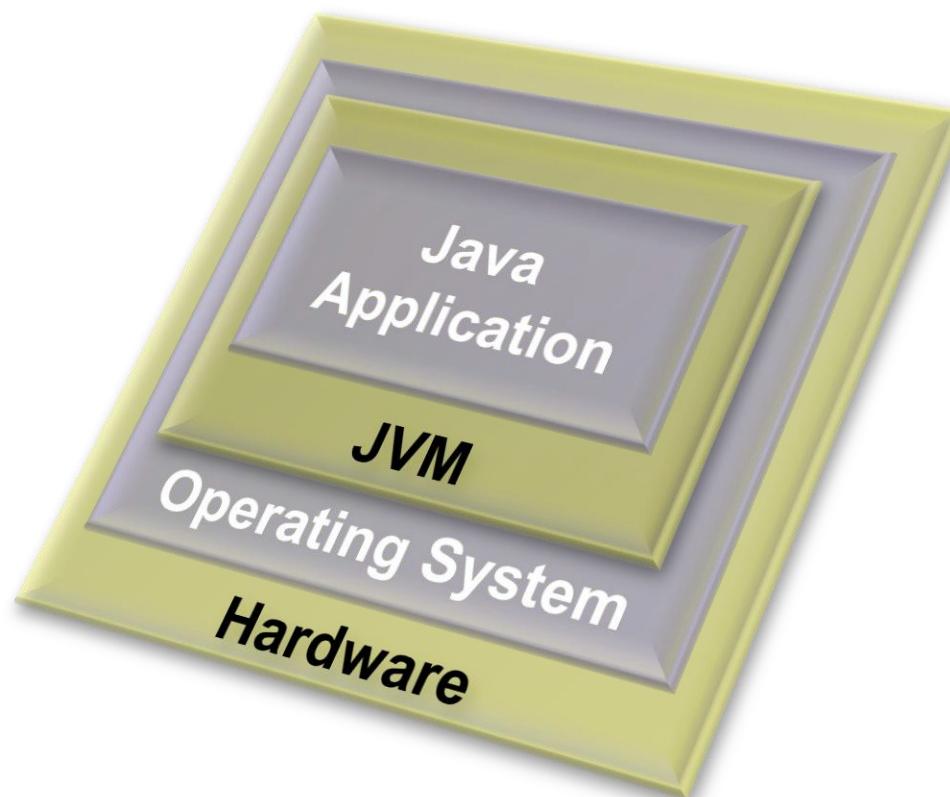
- Java was created by Sun Microsystems in **May 1995**.
- The Idea was to create a language for controlling any hardware, but it was too advanced.
- A team - **that was called the Green Team** - was assembled and lead by **James Gosling**.
- Platform and OS **Independent** Language.
- **Free** License;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

- 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 (Sun SPOT- Raspberry Pi)

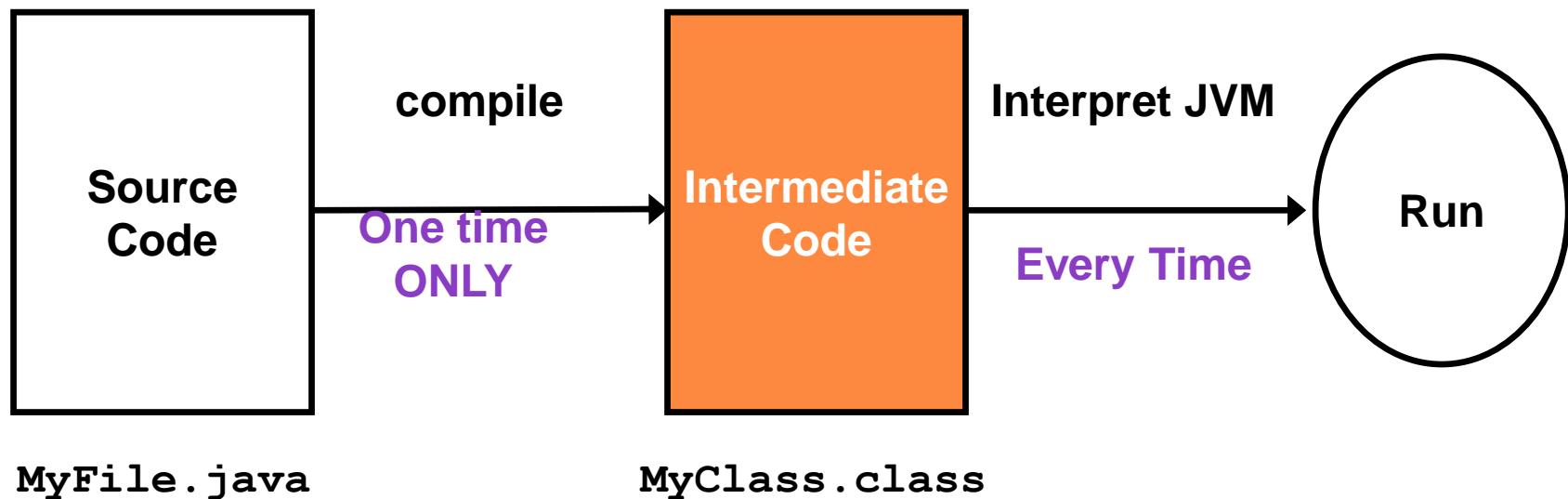
- Primary goals in the design of the Java programming language:
  - **Simple, object oriented, and easy to learn.**
  - **Robust and Secure.**
  - **Architecture neutral and portable.**
  - **Compiled and Interpreted.**
  - **Multithreaded.**
  - **Networked.**

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

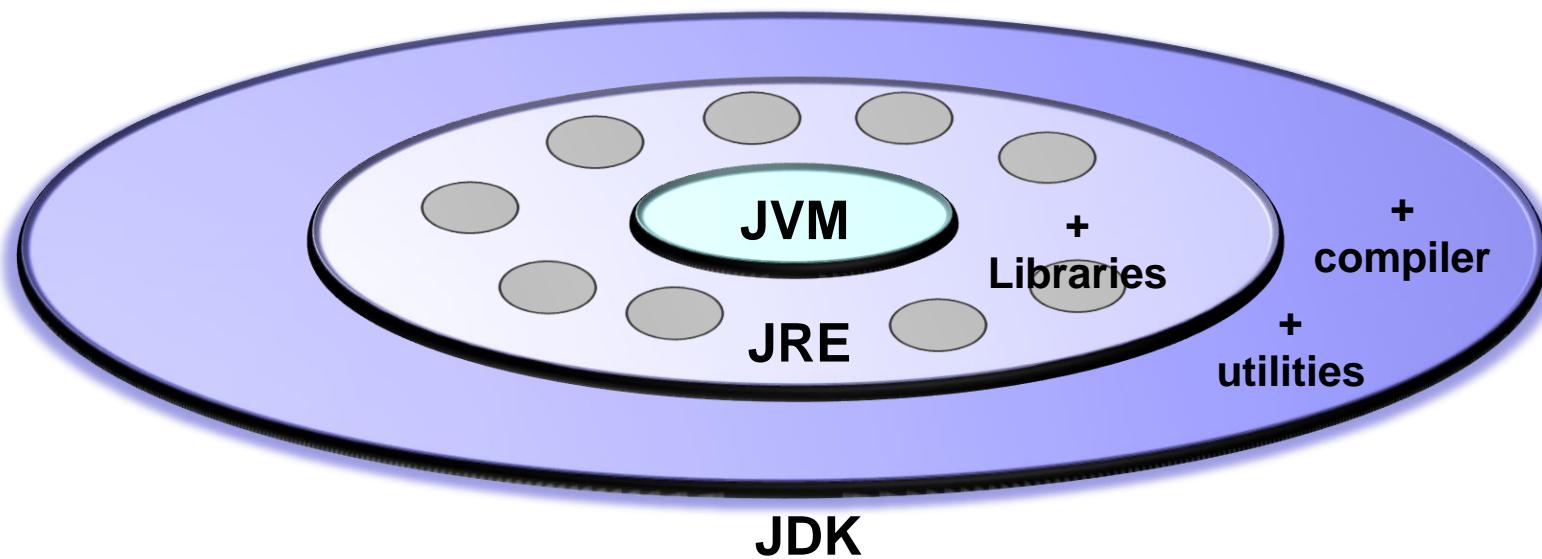
- Machine and Platform Independent



- Java is both, compiled and interpreted



- Java depends on dynamic linking of libraries



## Java development Kit (JDK)

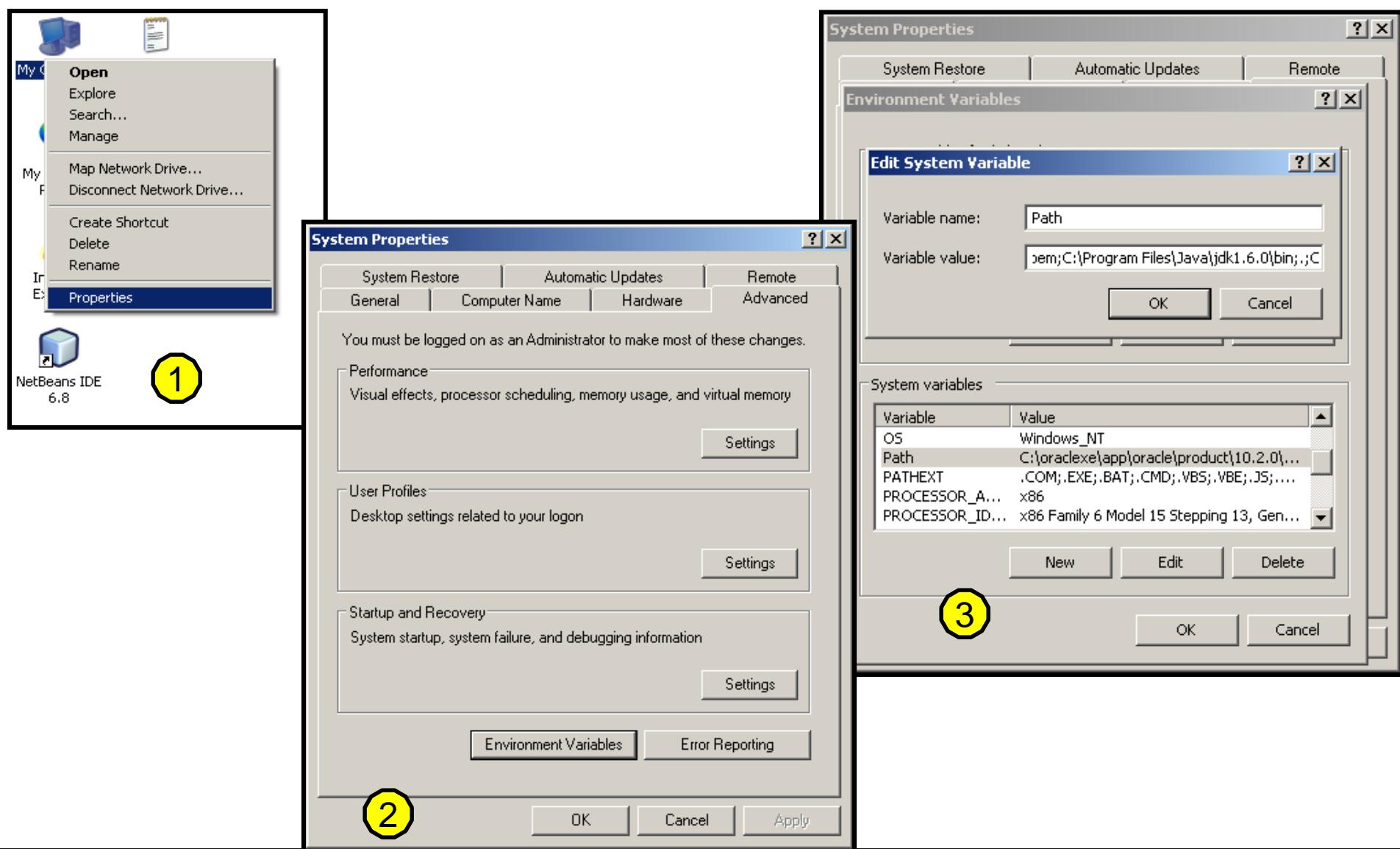
- Java is fully Object Oriented
  - Made up of Classes.
  - No multiple Inheritance.
- Java is a multithreaded language
  - You can create programs that run multiple threads of execution in parallel.
    - Ex: GUI thread, Event Handling thread, GC thread
- Java is networked
  - Predefined classes are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

- Download JDK 8

<http://bit.ly/3TJqjJA>

- 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\**
    - Open '**Control Panel**' then '**Advanced System Settings**'
    - 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



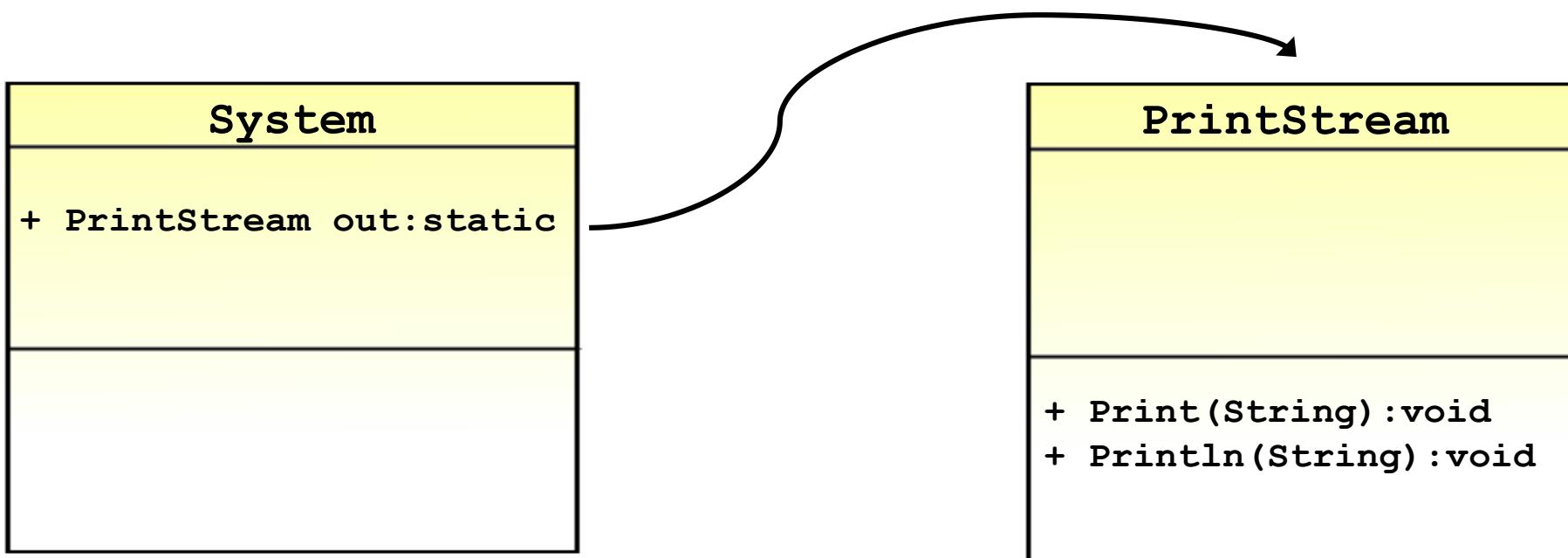
```
class HelloWorld  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello Java");  
    }  
}
```

**File name:** `hello.java`

- 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).



- **To compile:**

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

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

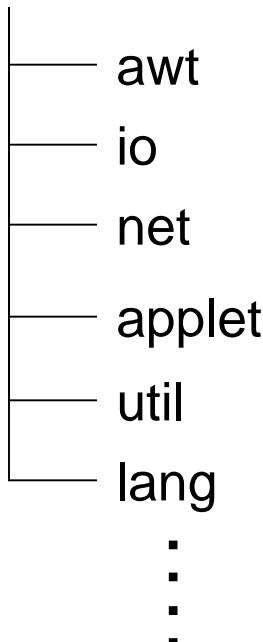
- **To run:**

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

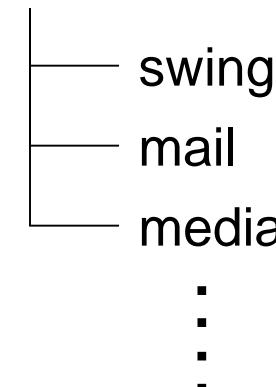
- Classes are placed in packages.
- We must import any classes that we will use inside our application.
- Classes that exist in package `java.lang` are imported by default.
- Any Class by default extends `Object` class.

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

java



javax



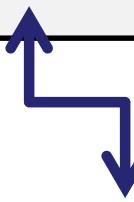
- If no package is specified,
  - then the compiler places the .class file in the default package (i.e. the same folder of the .java file).
- To specify a package for your application,
  - write the following line of code at the beginning of your class:

**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
```

- 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.

- To create a compressed JAR file:

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

- Example:

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

- 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>
```

Or without manifest file:

```
prompt>jar cef <entry-point> <archive_name.jar>  
<files>
```

# 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`



# Applets

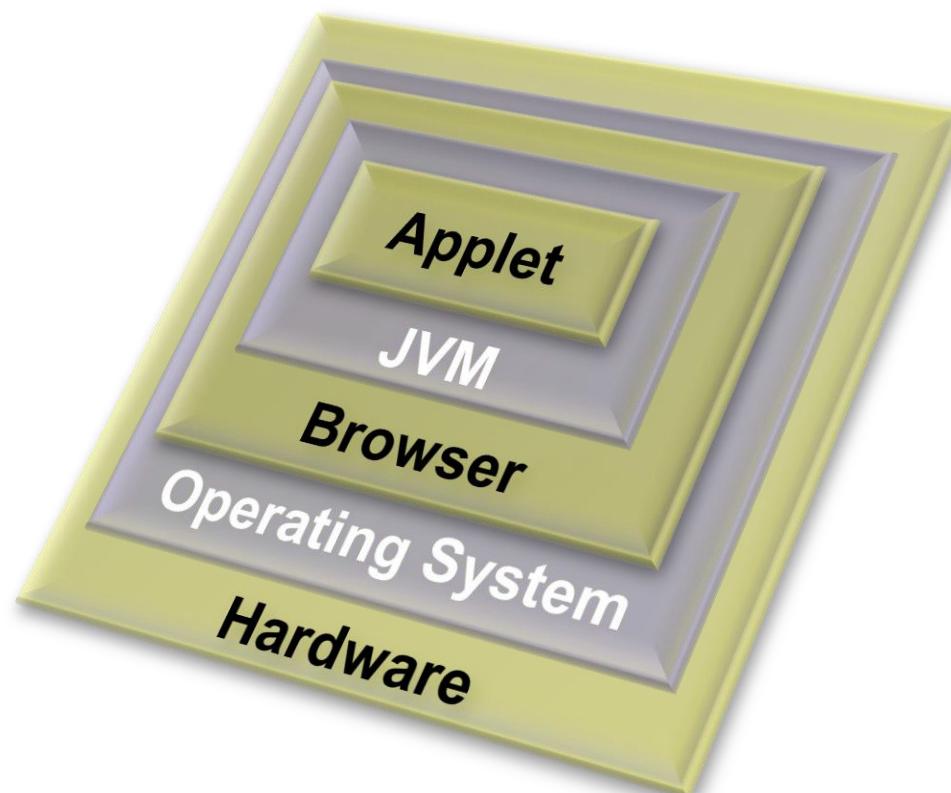
## Web Server (www.abc.com)



Download  
MyApplet.class



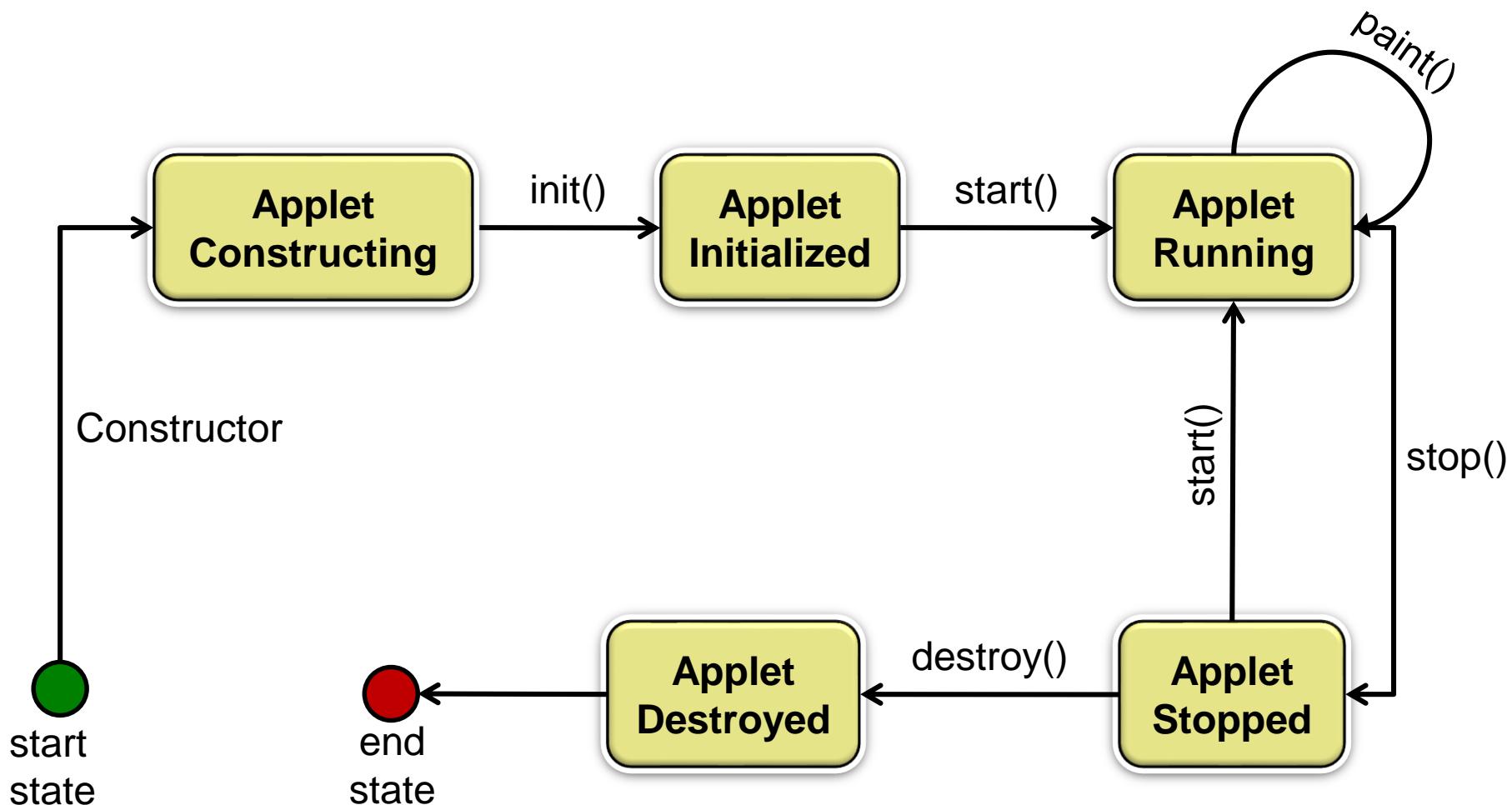
- Machine and Platform Independent



- 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.

- 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



- **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.

- 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.

```
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.

- 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.

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

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

- 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. 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. 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 2

## Introduction to OOP

## Using Java

- **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.
- Objects can share some of its *characteristics* with each other, but each object is unique by it self.
- Objects can interact with each other to accomplish *tasks*.

- **What is OOP?**

- Each object is an instance 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.

- **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).

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

**class**



**object**

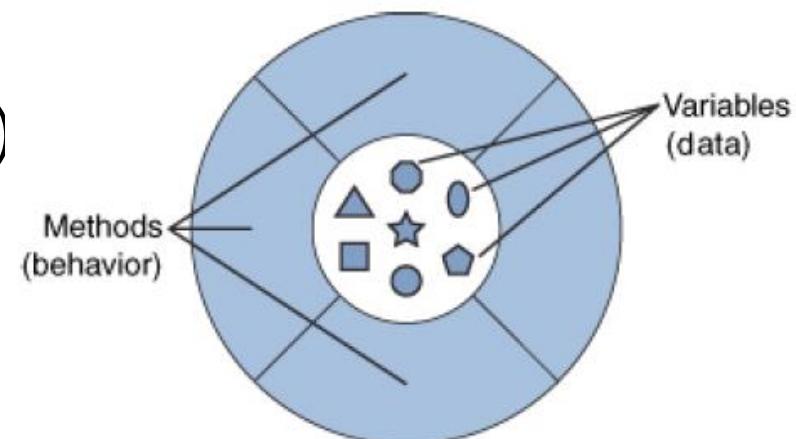


- **What is an Object?**

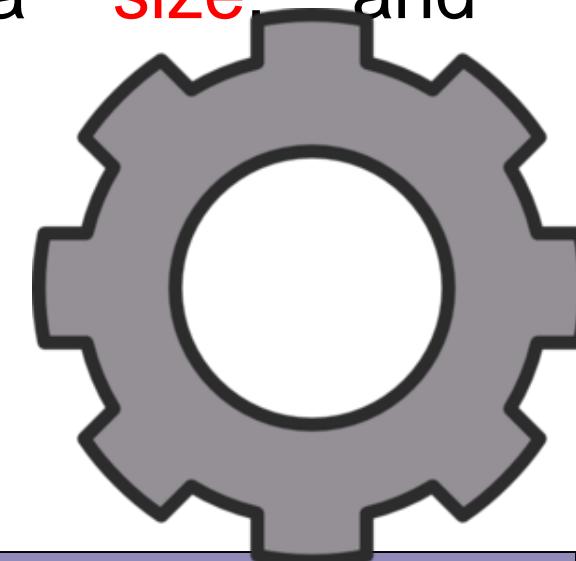
- An object is a software bundle of variables and related methods.

- Object consists of:

- **Data** (object's Attributes)
- **Behavior** (object's methods)



- Assume we have this object [SerratedDisc].
- How can we describe its Class?
  - First it's a serrated disc lets give it that name SerratedDisc.
  - Second each disk has a size and numberOfPins.
  - Last thing it can spin.



- So we have two different concepts here are very important.

Class	Object
Encapsulates the attributes, and behaviors into a blue-print code to provide the design concept.	The living thing that interacts and actually runs. 
Ex: class <b>SerratedDisc</b> { size numberOfPins spin }	<b>SerratedDisc</b> serr_Car = new <b>SerratedDisc</b> ();  size = 10 numberOfPins = 8

# How to create a class?

- To define a class, we write:

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

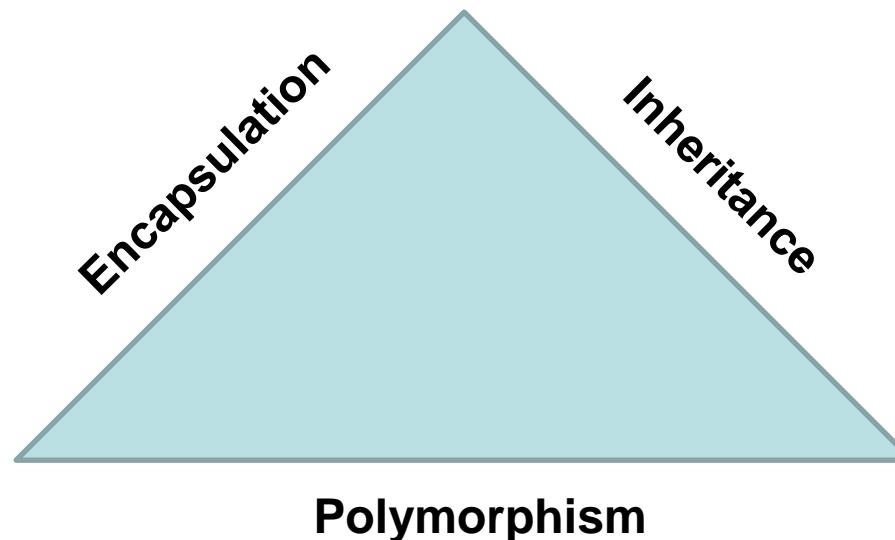
- Example:

```
public class SerratedDisc {
    private int size;
    private int numberOfPins;

    public void spin() {
        System.out.println("The disc is spenning now . . . ");
    }
}
```

# OOP Principles

- Object Oriented has main three principles:



- It is to encapsulate the data and behaviors in one class.
- In addition it is a language mechanism for restricting access to some of the object's components.
- Data hiding is done in Java using the **public**, and **private** key words.

```
public class SerratedDisc {  
    private int size;  
    private int numberOfPins;  
  
    public void spin(){  
        System.out.println("The disc is spenning now . . . ");  
    }  
}
```

private class members can only  
be accessed **within the class**

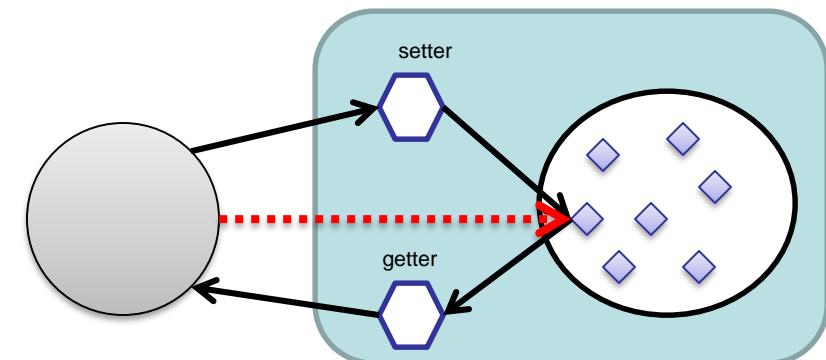
Public class members can be accessed  
within the class or **from another class**

# Encapsulation cont'd

```
public class SerratedDisc {  
    private int size;  
    private int numberOfPins;  
  
    public void spin(){  
        System.out.println(size); ← Accessible within same class  
        System.out.println("The disc is spenning now . . . ");  
    }  
}
```

```
class Main{  
    size has private access in SerratedDisc  
    ----  
    (Alt-Enter shows hints)    void main(String[] args){  
        sc myCarDisc = new SerratedDisc();  
        myCarDisc.size = 10; ← Cannot be accessed outside the  
        myCarDisc.spin();       class  
    }  
}
```

- As we can see in the previous example private class members are not accessible outside the class.
- These keywords are called access modifiers, and it can be added to attributes and methods.
- The point of encapsulation is to hide class attributes from the outside world so other objects can not access them directly, but to let class methods to be an interface to access them.



- Exposing object attributes might lead to misuse of this attributes.
- Instead of exposing object attributes OOP uses the setter and getter methods.

```
public int getSize() {  
    return size;  
}  
  
public void setSize(int size) {  
    this.size = size;  
}  
  
public int getNumberOfPins() {  
    return numberOfPins;  
}  
  
public void setNumberOfPins(int numberOfPins) {  
    this.numberOfPins = numberOfPins;  
}
```

- Now we can access the object attribute through the setter and getter method like the following:

```
class Main{  
    public static void main(String[] args) {  
        SerratedDisc myCarDisc = new SerratedDisc();  
        myCarDisc.setSize(10);  
        myCarDisc.spin();  
    }  
}
```

- 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

- To declare a method:

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

- Example:

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

- 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.
  - 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.

- To make use of this class we need to **construct** an object of this class.

```
SerratedDisc serr_Car = new SerratedDisc();
```

- Constructing an object is like building it and making it ready for action inside the program.
- To construct an object we use a special type of methods called **Constructor**
- A constructor is simply a public method that does not have a return and its name matches the class name (case sensitive).

- By default any class - we write - has a constructor without the need to write it, called **default constructor**.
- The default constructor does nothing but it must be called before creating an object.
- The constructor is the best place to put initialization per-object.
- If no constructors are written then the compiler uses the class' default constructor.
- If we wrote a custom constructor that takes arguments then there will not be a default constructor.

- To construct an object of class we use the `new` keyword.

```
class Main{  
    public static void main(String[] args){  
        SerratedDisc myCarDisc = new SerratedDisc();  The disc is created  
        myCarDisc.spin();   Asking the disc to spin  
    }  
}
```

- The output of this program will be like

```
: Output - SerratedDisc (run)  
run:  
The Disc is spenning now!  
BUILD SUCCESSFUL (total time: 0 seconds)
```

- To add a constructor to our class we need to code extra method like - with the same name of the class (case sensitive).

```
public class SerratedDisc {  
    private int size;  
    private int numberOfPins;  
  
    public void spin(){  
        System.out.println("The disc is spenning now . . . ");  
    }  
  
    public SerratedDisc (int size, int numberOfPins){  
        this.size = size;  
        this.numberOfPins = numberOfPins;  
    }  
}
```

- Now to construct an object we need to use the new constructor.

```
class Main{  
    public static void main(String[] args){  
        //SerratedDisc myCarDisc = new SerratedDisc();  
        SerratedDisc myCarDisc = new SerratedDisc(10, 6);  
        myCarDisc.spin();  
    }  
}
```

- Now using the default constructor is illegal because we have our constructor and the default constructor as if it has never existed.

- Any attribute or method are declared inside the class body are called instance variable, or method.
- For each new object is created and a new location for its instance attributes is located inside the memory.
- Methods are located once.
- When a method is called from an object, a reference points to the that object would be passed to the method implicitly. Such reference is called **this** reference.
- Using the **this** reference is not obligatory within the class.

# Instance and Static Members cont'd

- There is some cases we must use the **this** reference, like:

```
public SerratedDisc (int size, int numberOfPins){  
    this.size = size;  
    this.numberOfPins = numberOfPins;  
}
```

- Attributes and methods can be declared **static**.
- Static attributes do not belong to a single instance of this class. They belong to the class itself and could act as shared resources to all instances.
- Static variable are located only once inside memory before the creation of any object.

- Static variables can be referenced at any method inside the class (instance, or static).
- Static methods can only access the static variables of the class.
- Public static variables, or methods can be accessed outside the class using the class name without the need to construct new object.

```
public class Main {  
  
    public static void main(String [] args){  
        //SerratedDisk myCarDisk = new SerratedDisk();  
        SerratedDisk.getManufacturerName();  
        //SerratedDisk myCarDisk = new SerratedDisk(10, 6);  
        //myCarDisk.spin();  
  
    }  
}
```

# Instance and Static Members cont'd

```
public class SerratedDisc {  
    private int size;  
    private int numberOfPins;  
    private static String MANUFACTURER_NAME;  
  
    public void spin(){  
        System.out.println("The disc is spenning now . . . ");  
        System.out.println("The Disc's manufacturer name is: " +  
                           SerratedDisc.MANUFACTURER_NAME);  
    }  
    public static void setManufacturerName(String name){  
        MANUFACTURER_NAME = name;  
    }  
    public static String getManufacturerName(){  
        return MANUFACTURER_NAME;  
    }  
    public SerratedDisc (int size, int numberOfPins){  
        this.size = size;  
        this.numberOfPins = numberOfPins;  
    }  
}
```

# Lab Assignments

## 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:  $x + yi$**  ,  **$5+6i$**

### 3. Simple Prompt Application

- Create a simple non-GUI Application that represent Student and has method to print student info. :

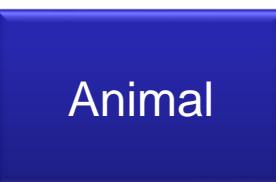
**Student :** name, age, track.....

# Lesson 3

## OOP II and Applet

- **Inheritance:**
  - Inheritance is to extend the functionality of an existing class
  - Inheritance represents **is – a** relationship between the Child and its Parent.
- **Using inheritance comes with a set of benefits:**
  - Reduces the amount of code needed to be written by developers.
  - Makes the code easy to maintain and update.
  - Helps in building more reasonable class hierarchy and simulating to the real world.

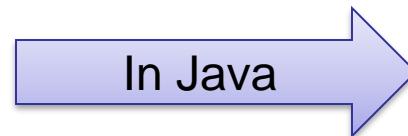
# Inheritance



Kind of  
or  
is - a

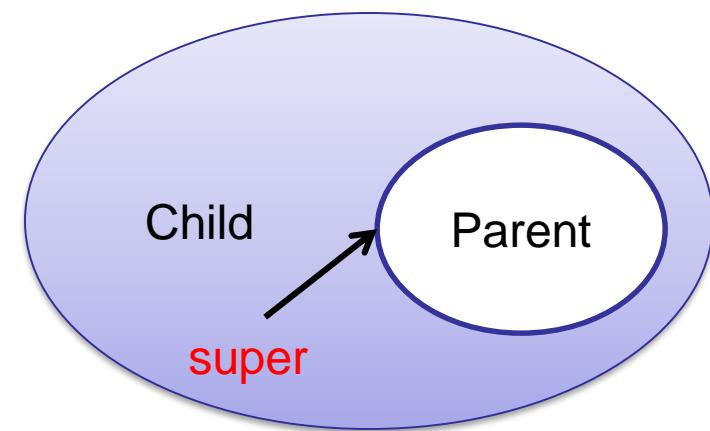


```
public class Animal {  
  
    private String name;  
    private int numOfLegs;  
    private float speed;  
  
    public void play(){  
        System.out.println("the animal is playing.....");  
    }  
}
```



```
public class Dog extends Animal{ ← Inheritance  
  
    public void woof(){  
        System.out.println("Dog: Woof Woof....");  
    }  
}
```

- the super reference can only point to parent public members.



- In the animal example we want to be able to access the private members of the parent class without the need to violate the encapsulation rule.

- We can mark our attributes and methods as **protected** to indicate that it is accessible to child classes but not to the outside world.

```
public class Animal {  
  
    protected String name;  
    protected int numOfLegs;  
    protected float speed; ←  
  
    public void play() {  
        System.out.println("the animal is playing.....");  
    }  
}
```

Now all of this attributes are accessible inside the Animal class or any sub class of it, but not accessible to any class outside that hierarchy

- As polymorphism can be seen in inheritance it also can be seen in class attributes and methods.
- The first polymorphism concept that can be seen inside one class or more than one class with inheritance relationship is the **overloading**.
- Overloading is re-writing a method with a name already defined for another method inside the same class or a parent, but with changing the method parameter section.
  - Changing method parameter types.
  - Changing method parameter order.
  - Changing method parameter count.
- Note: **changing the method return type is not considered overloading and will give compilation error**

# Polymorphism (overloading)

```
public void draw(String s) {  
  
}  
public void draw(int i) {  
  
}  
public void draw(double f) {  
  
}  
public void draw(int i, double f) {  
  
}  
public int draw(int i, double f) {  
    return 5;  
}
```



Compilation error

**Note:** attributes cannot be overloaded.

# polymorphism

- Second type of polymorphism is **overriding**.
- Overriding is re-writing a method typically as it was written in its parent class inside a child class.
- Overriding can only happen inside different classes inside the same class hierarchy structure.

```
public class Animal {                                public class Dog extends Animal{  
  
    private String name;  
    private int numOfLegs;  
    private float speed;  
  
    public void play(){  
        System.out.println("the animal is playing.....");  
    }  
}  
  
→  
public void woof(){  
    System.out.println("Dog: Woof Woof....");  
}
```

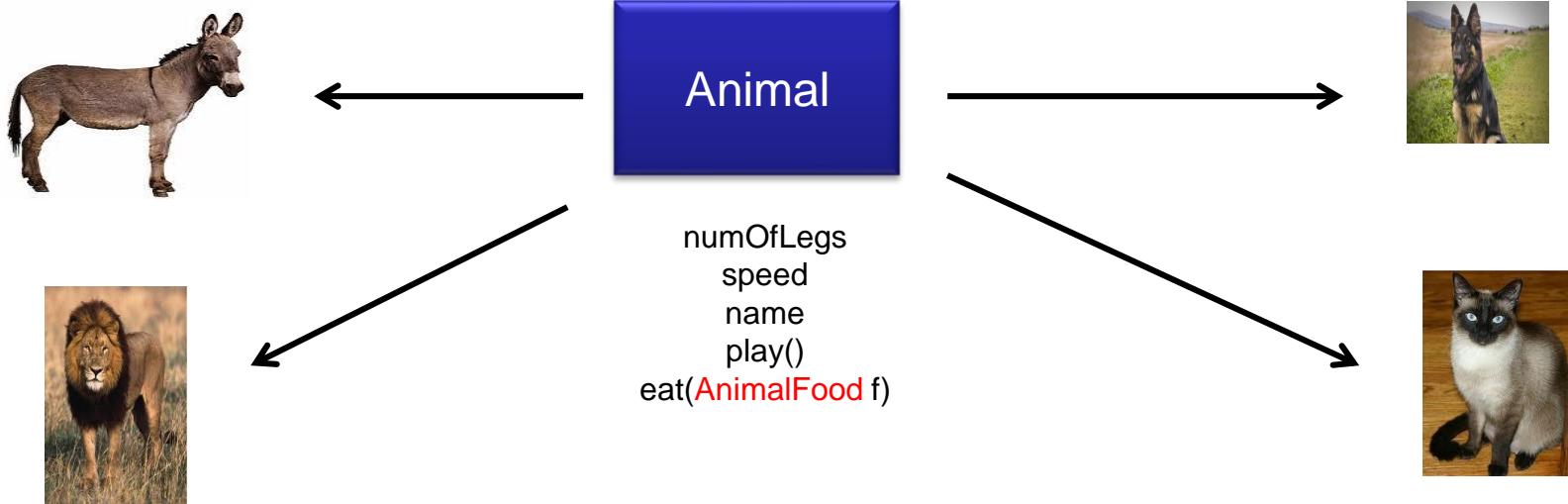
- Now if we try to call the play method inside the Dog class the newly overridden method inside the Dog class will respond.
- If we want to call the parent method we can use the super reference.
- Overloading and overriding are used to redefine the method implementation with preserving the method name.
- This helps the class users to remember only a little about my class, and in the other hand the class will behave differently according to the implemented method.
- Note: **overriding can be done to attributes but it is not recommended.**

- Polymorphism allows that one reference type can point to different objects as long as they are in the correct class hierarchy.

```
public class Main {  
  
    public static void main(String[] args) {  
        Animal myDog = new Dog();  
        myDog.play();  
    }  
  
}
```

- And when calling the overridden method play() it will call the correct object method of the Dog class.

- Abstraction is the process by which a data and programs are defined with a presentation similar in form to its meaning while hiding away the implementation details.
- Abstraction is a concept or idea not attached to any instance.



- At this point the class Animal is not well defined and implementing its methods at this point will be pointless.
- Also instantiating an object of class Animal is also pointless as the Animal class is only a concept class to indicate that any animal can have these properties but it is only abstract definition.
- So it's better to declare the class Animal as abstract class, rather than declaring a normal class.
- This provides a more professional and efficient class hierarchy building.

- Abstract class is created by adding the abstract key word before the class name.

```
public abstract class Animal {
```

- At this point we cannot instantiate any objects from this class.
- Abstract class can have zero or more abstract methods.
- Abstract method is a method with no implementation body only the method signature.

```
abstract public void play();
```

- Abstract class can be inherited by other classes and its abstract methods overridden to provide an implementation to a case of this abstract class.
- Inheriting from abstract class without overriding its abstract methods makes your child class also abstract.
- Overriding the abstract method is a must or declare your child class as abstract too.
- Abstract class can have zero or more abstract methods, but abstract methods cannot exist in normal class.

# Lesson 4

## Data Types & Operators

- 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 can be classified into two types:

Primitive

Reference

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



- Each primitive data type has a corresponding wrapper class.

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

- 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).

- 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);
```

- They have special static representations, for example:

---

**POSITIVE\_INFINITY**

---

**NEGATIVE\_INFINITY**

---

**NaN**

Not a Number

In class `Float & Double`

---

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

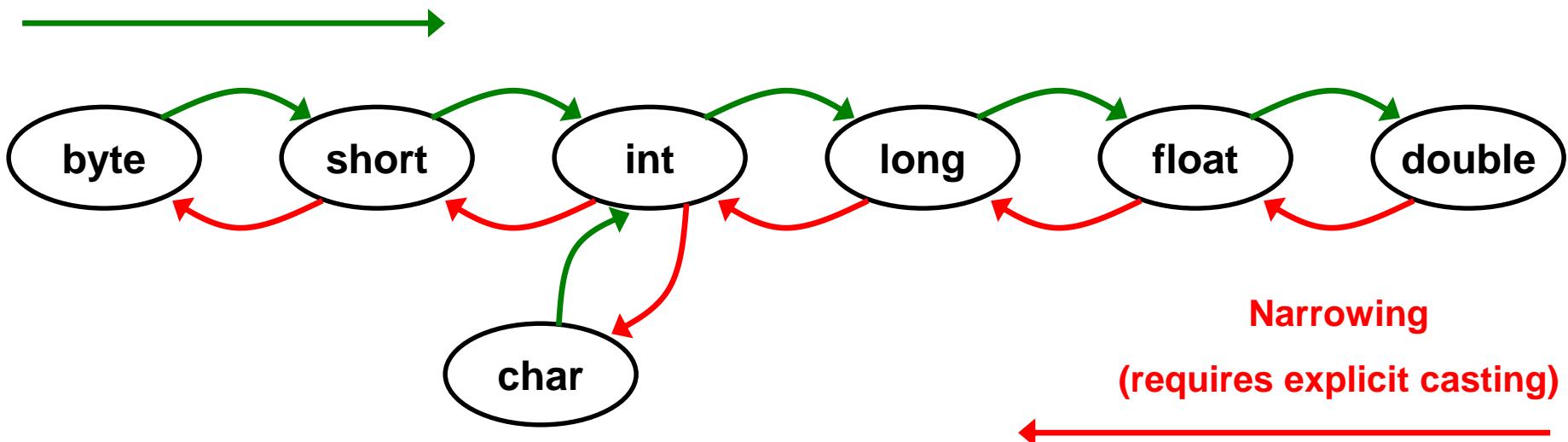
boolean	true false	
char	'a' .... 'z'	'A' .... 'Z'
	\u0000' .... '\uFFFF'	
	\n' '\r' '\t'	
Integral data type	15	Decimal (int)
	15L	Decimal (long)
	017	Octal
	0XF	Hexadecimal
Floating point data type	73.8	double
	73.8F	float
	5.4 E-70	5.4 * 10 <sup>-70</sup>
	5.4 e+70	5.4 * 10 <sup>70</sup>

- 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.

- Unary Operators:

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

**Widening**  
(implicit 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

- 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)

- Ternary Operator:

**condition ?true statement:false statement**

```
int y=15;
```

```
int z=12;
```

```
int x=y<z?10:11; 
```

```
If(y<z)  
x=10;  
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=

- 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

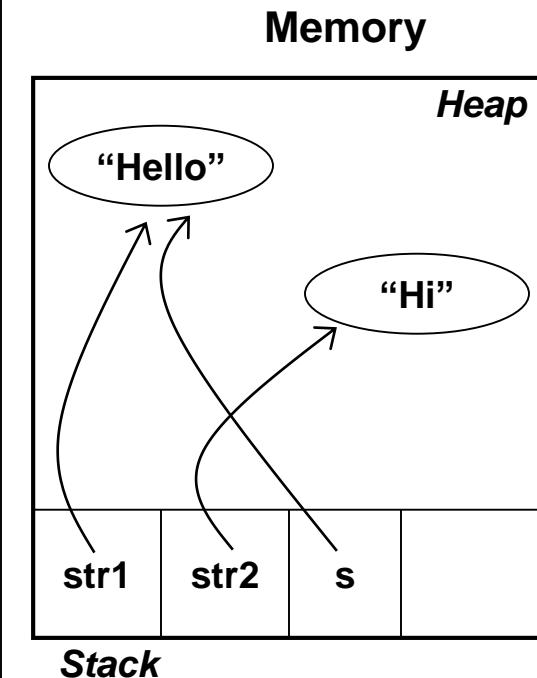
String str2 = new String("Hi");

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
```



# Lesson 5

## Using Arrays & Strings

- 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.

- 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 };
```

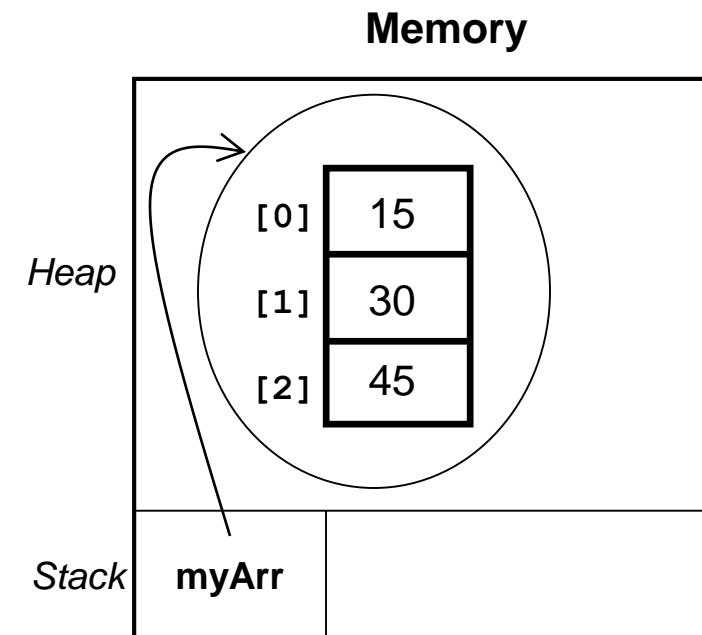
- To determine the size (number of elements) of an array at runtime, use:

```
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:

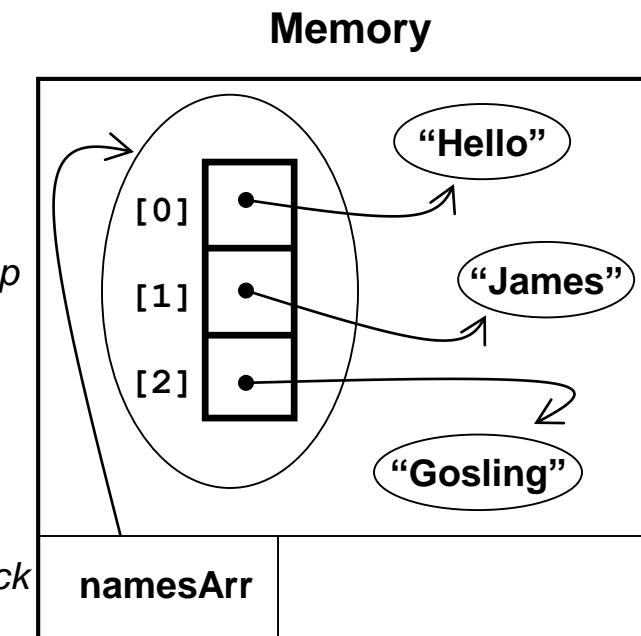
```
String[] namesArr;
```

```
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]);
```



- Although String is a reference data type (class),
  - it may figuratively be considered as the 9<sup>th</sup> 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)
```

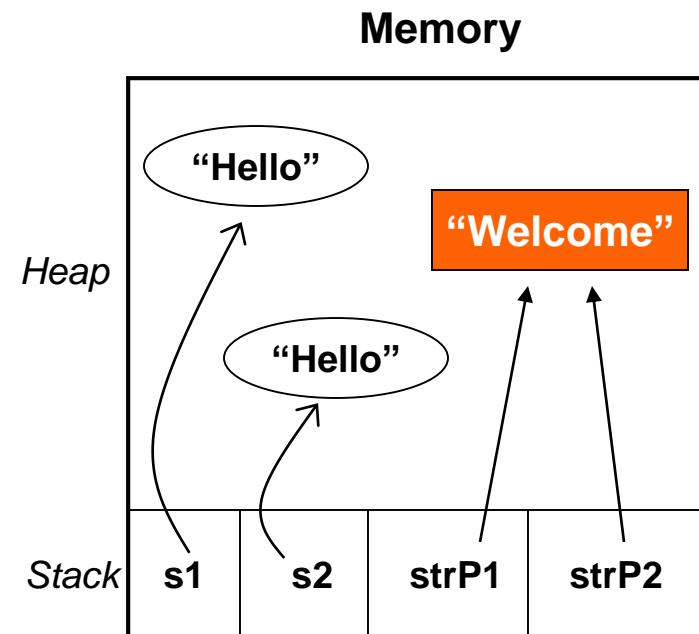
- The ‘+’ and ‘+=‘ operators were overloaded for class String to be used in concatenation.

```
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 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 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

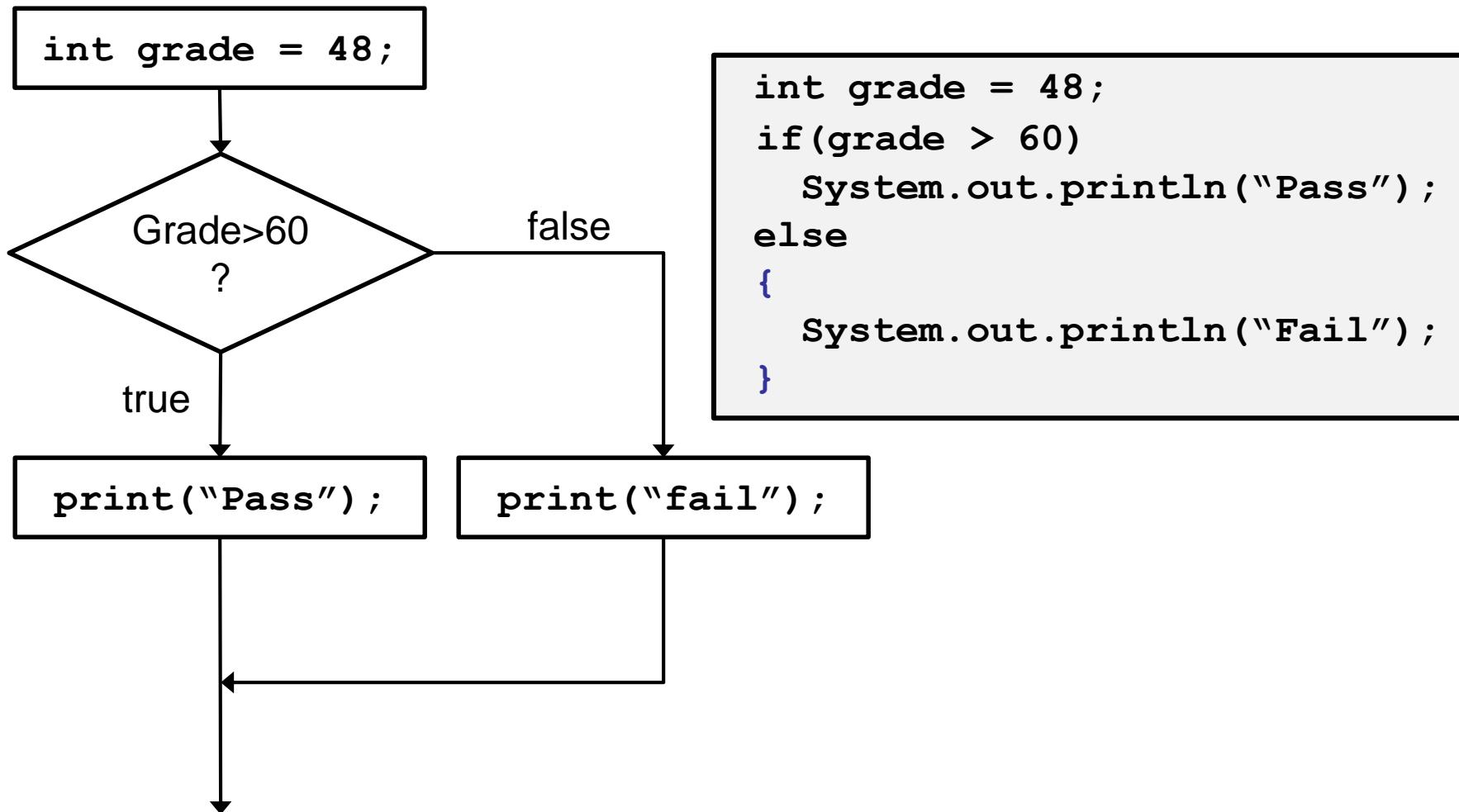
## Controlling Program Flow

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

```
if(boolean_expr)
{
    ...
    ...      //true statements
    ...
}

[else]
{
    ...
    ...      //false statements
    ...
}
```

# if, else Example



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

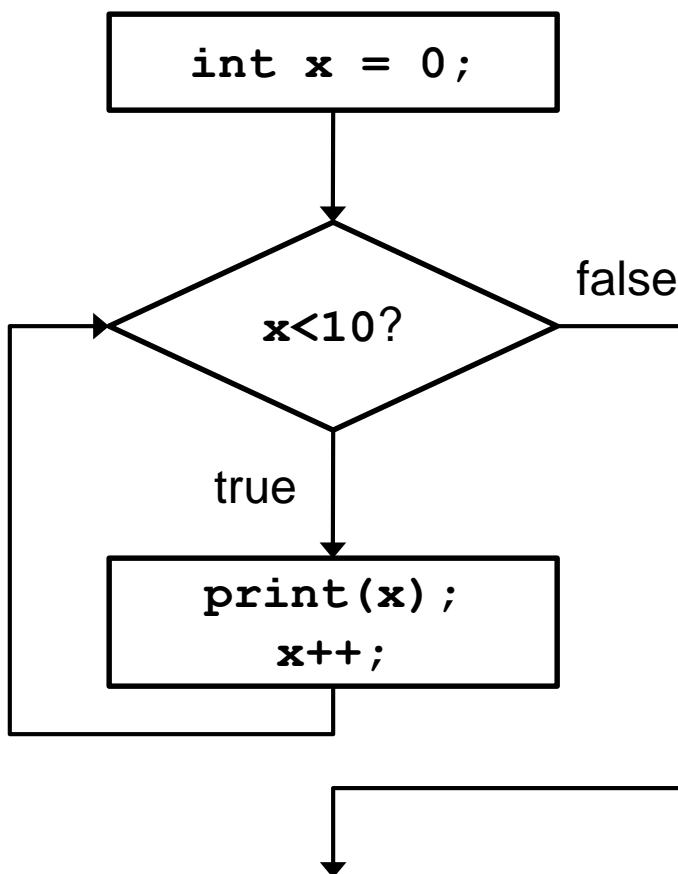
```
switch (myVariable) {  
    case value1:  
        ...  
        ...  
    break;  
    case value2:  
        ...  
        ...  
    break;  
    default:  
        ...  
}
```

- byte
- short
- int
- char
- enum
- String “Java 7”

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

```
while (boolean_condition)
{
    ...
    ...
    ...
}
```

# while loop Example

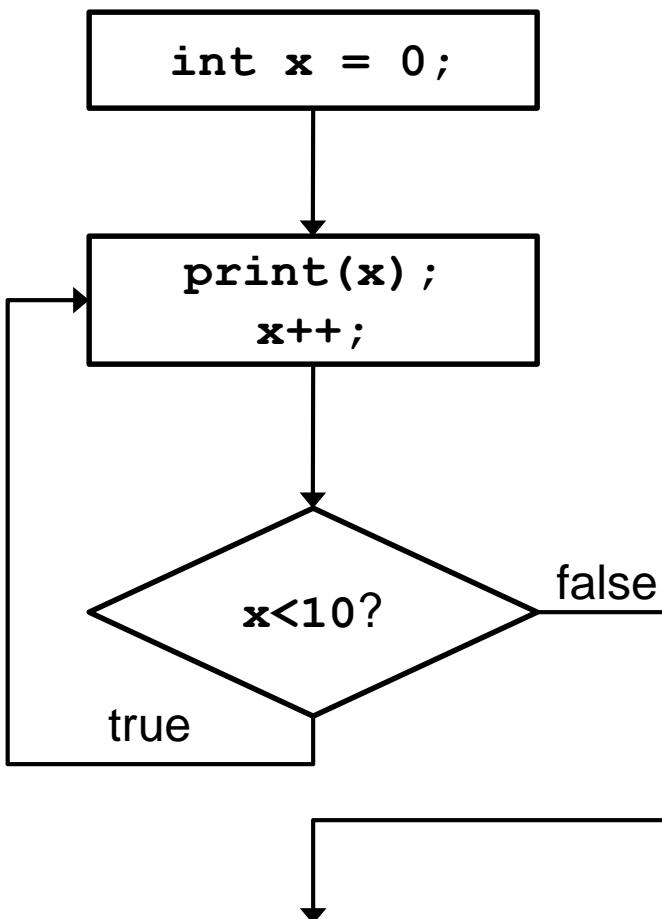


```
int x = 0;  
while (x<10) {  
    System.out.println(x);  
    x++;  
}
```

- 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);
```

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

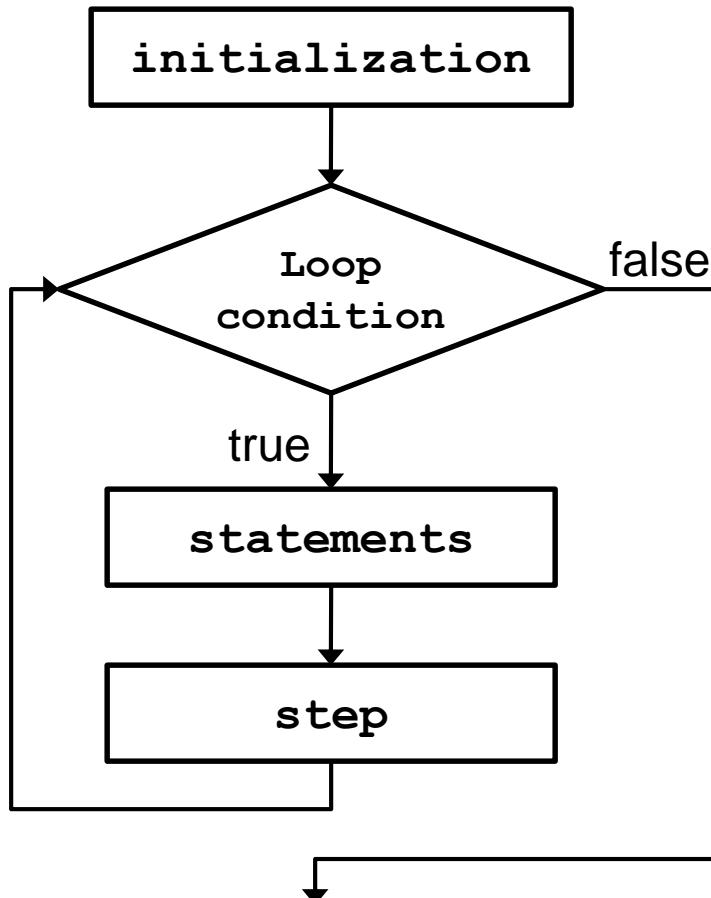
```
for (initialization ; loop_condition ; step)
{
    ...
    ...
    ...
}
```

```
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)
```



```
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.

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

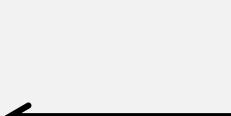
```
double average = 0.0;  
for(int i=0;i<samples.length;i++)  
{  
    average += samples[i];  
}  
  
average /= samples.length;
```

```
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 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;
}
.....
```



- To comment a single line:

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

- To comment multiple lines:

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

- 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
```

- 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 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

- Using: 1. split    2. StringTokenizer    3. substring & indexOf

- Write a program that print the following patterns:  
using 2 loops:

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

- **Bonus:** Write the same program using 1 loop

# Lesson 6

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

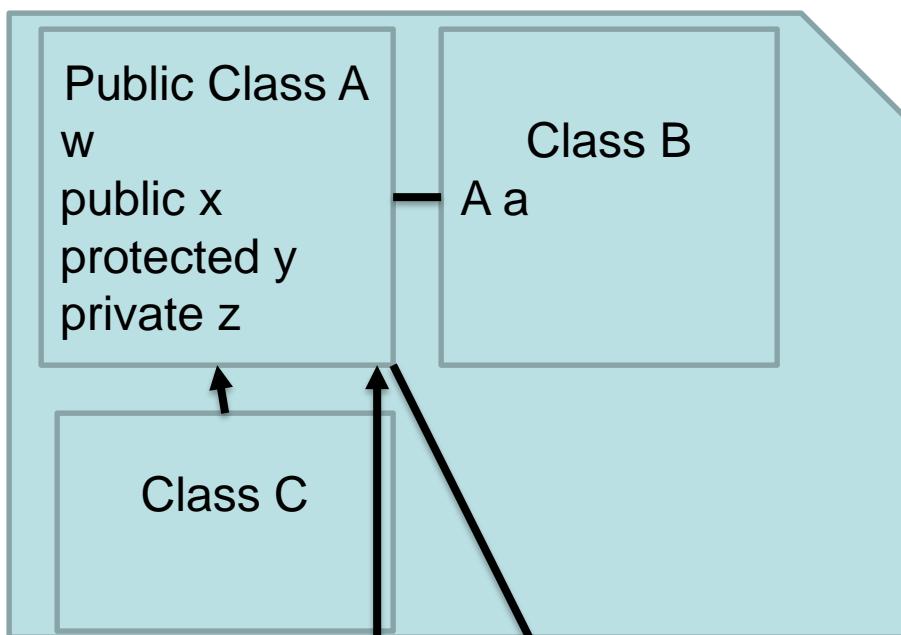
- Modifiers and Access Specifiers
- Graphics, Color, and Font Class
- Exception Handling

- 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.

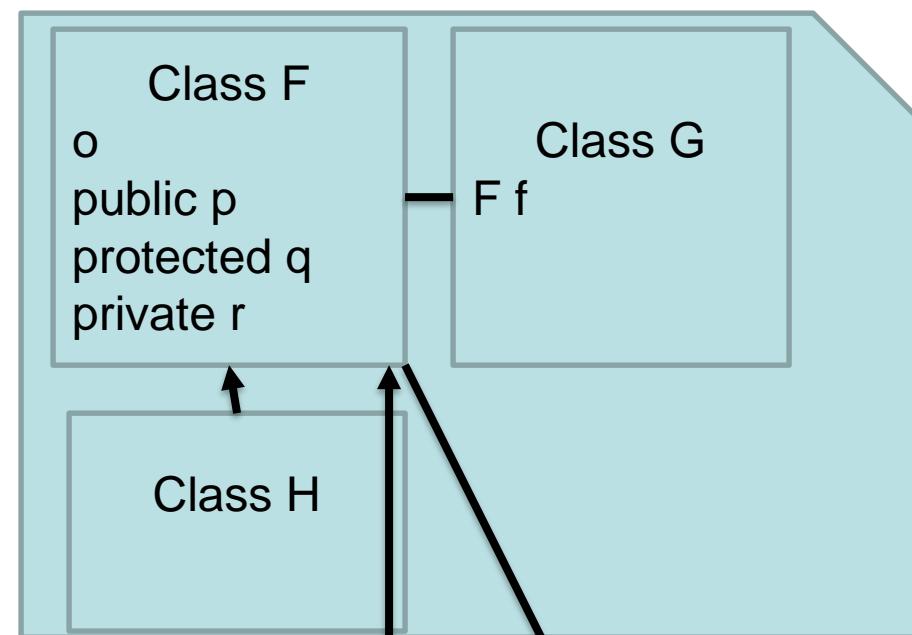
# Modifiers and Access Specifiers cont'd

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

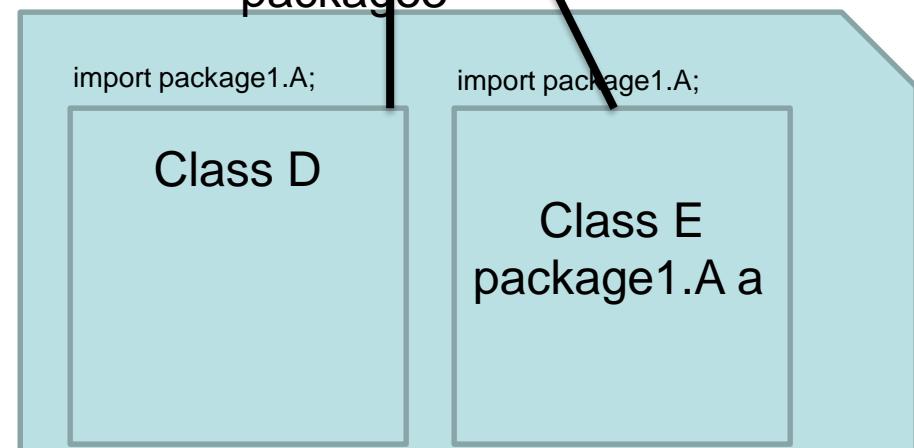
package1



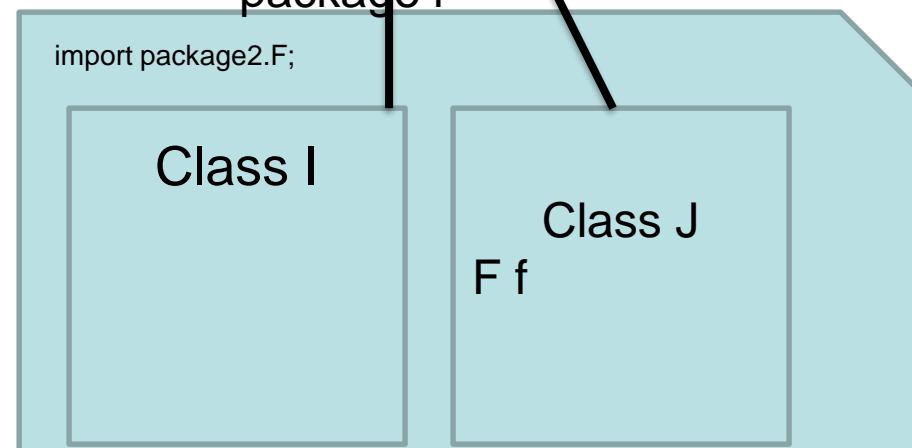
package2



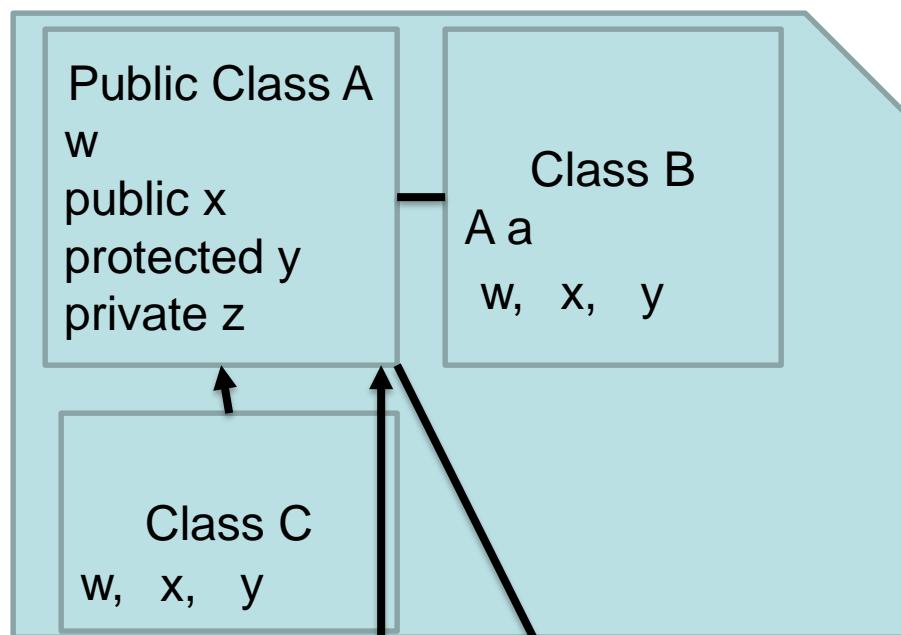
package3



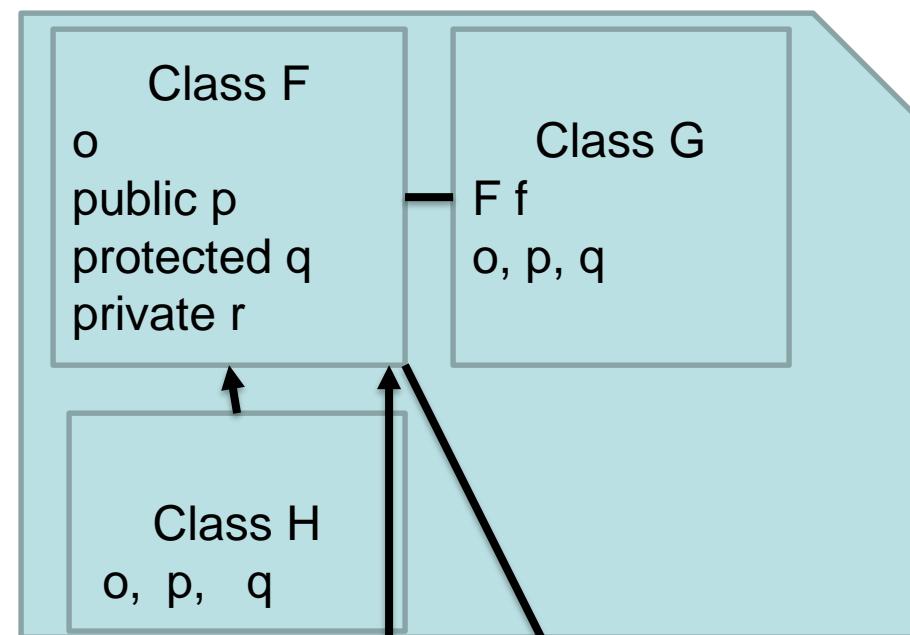
package4



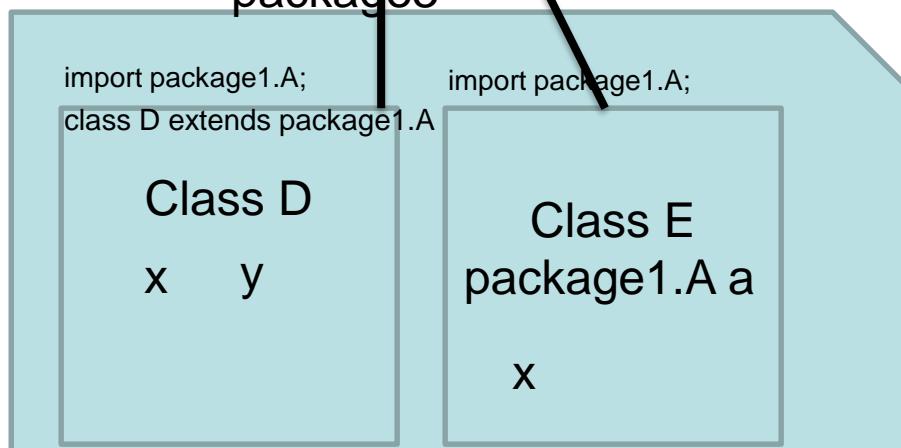
package1



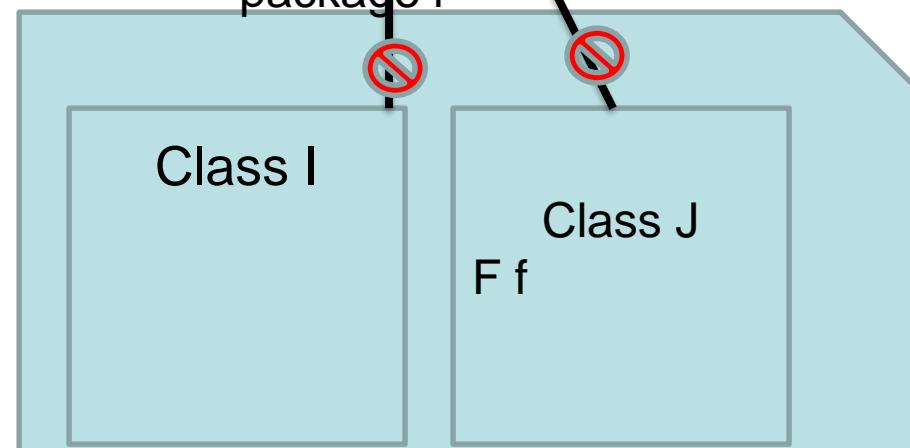
package2



package3



package4



- 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)`

- 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);
```

- 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.

- 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`

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

- 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:

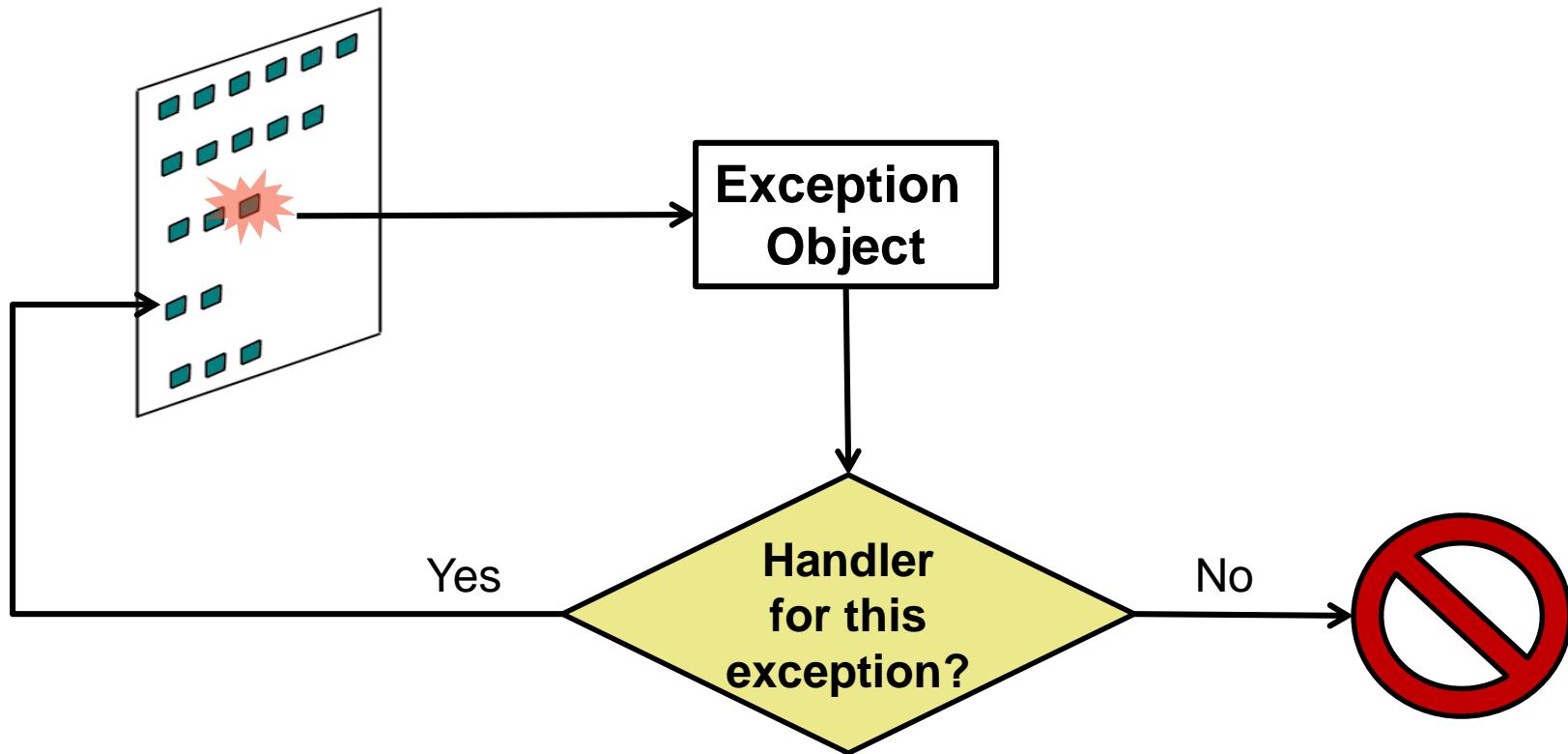
```
Toolkit t= Toolkit.getDefaultToolkit();  
String[] s = t.getFontList();
```

- Objects of class Font are immutable.

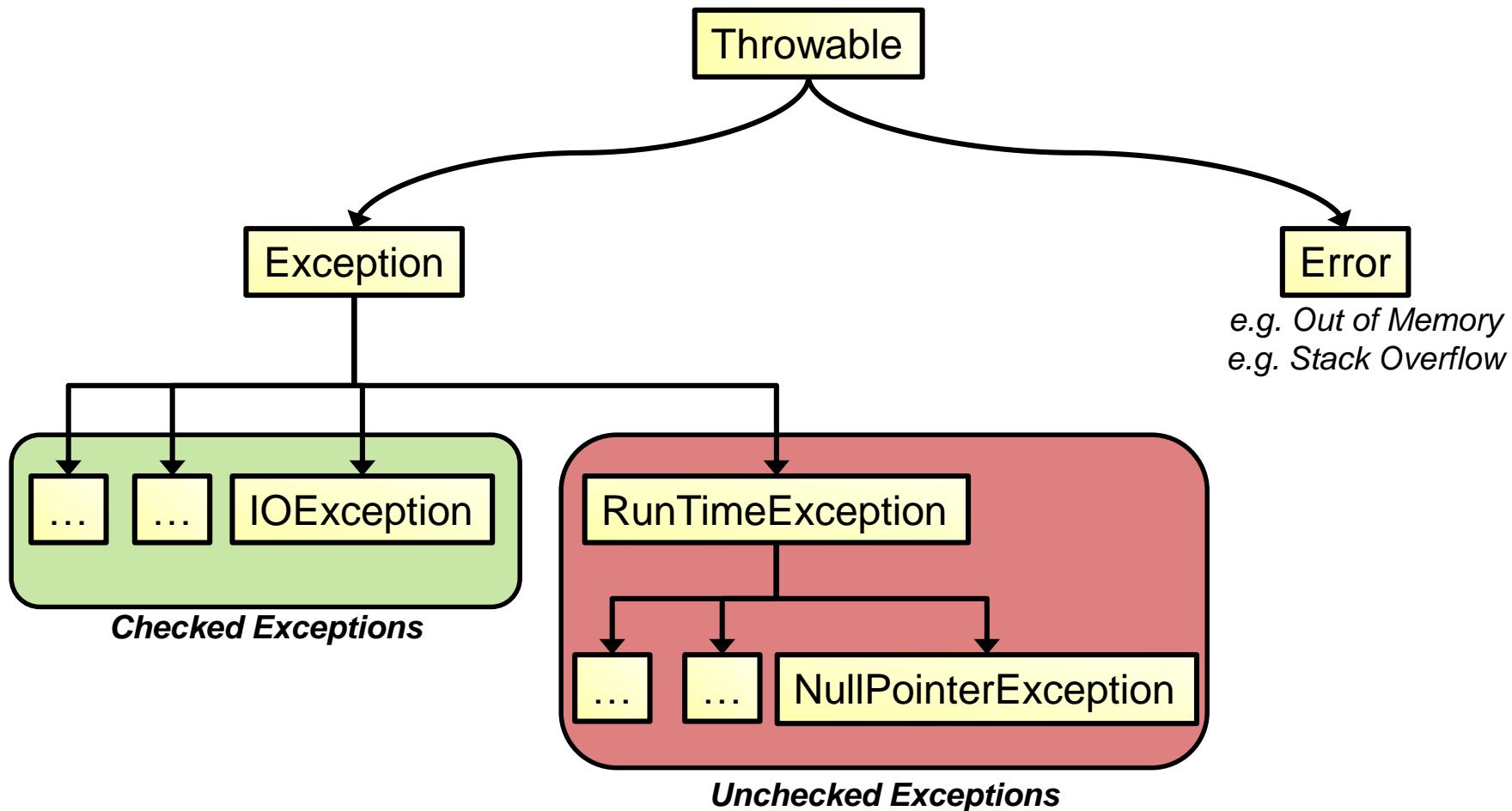
- An exception is an object that's created when an abnormal situation arises in your program **during runtime.**
- **Example:**
  - attempting to open a file that does not exist, or
  - attempting to write in a file that the OS has marked as read only.
  - attempting to use a reference whose value is null, or
  - attempting to access an array element that is beyond the actual size of the array.

- The exception object has description about the nature of the problem.
- The exception is said to be *thrown* when the problem occurs
- The code receiving the exception object as a parameter is said to *catch it*.

# How Does Java Handle Exceptions?



# Exceptions cont'd



- In order to deal with an exception,
  1. Catch the exception and handle it by include three kinds of code blocks: try, catch, and finally.
  2. Let the exception pass to the calling method by declare the method to throw the same exception.
  3. Catch the exception and throw a different exception.

1. Including three kinds of code blocks to handle them: try, catch, and finally.

- **The try block:**

encloses the code that may throw one or more exceptions.

- **The catch block:**

encloses code that handles exceptions of a particular type that may be thrown in the associated try block.

- **The finally block:**

is used to write code that will always definitely be executed before the method ends, whether the exception occurs or not.

# Handling Exceptions

```
public class Test{  
    public void testMethod() {  
        MyClass m = new MyClass();  
        try{  
            ...  
            m.myMethod(7) ; //a method that throws an exception  
            ...  
        }  
        catch(SomeException ex) {  
            //handle the exception here  
            ex.printStackTrace() ; //print details of the exception  
        }  
    }  
}
```

- Moreover, you could also use a **finally** block:

```
public class Test{  
    public void testMethod() {  
        MyClass m = new MyClass();  
        try{  
            ...  
        }  
        catch(XYZException ex) {  
            //handle the exception here  
        }  
        finally{  
            //the code that will always be executed  
        }  
    }  
}
```

2. declare the method to throw the same exception:

```
public class Test{
    public void testMethod() throws XYZException
    {
        MyClass m = new MyClass();
        m.myMethod(7);
    }
}
```

# Throwing an Exception

- Now let's take a closer look at **MyClass** and **readData** method to see how an exception is created and thrown:

```
public class MyClass{
    public String readData() throws EOFException{
        ...
        if(n < length)
            throw new EOFException();
        return s;
    }
}
```

- 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**.

- 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.

```
try { ..... }

catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: ");
}

catch (IOException e) {
    System.err.println("Caught IOException: ");
}
```

## In Java 7:

```
try { ..... }

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

- 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 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();
}
}
```

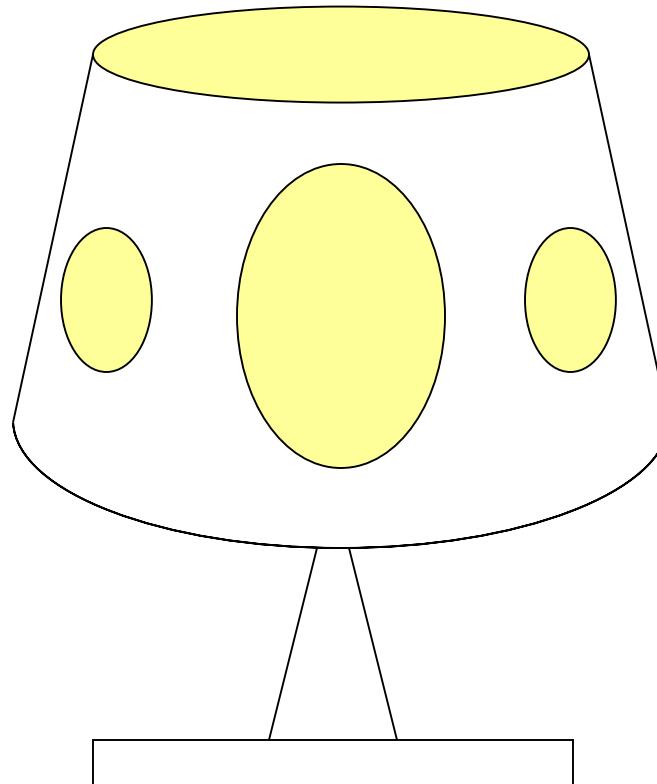
# Lab Exercise

# 1. 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).

## 2. Drawing a Lamp on Applet

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



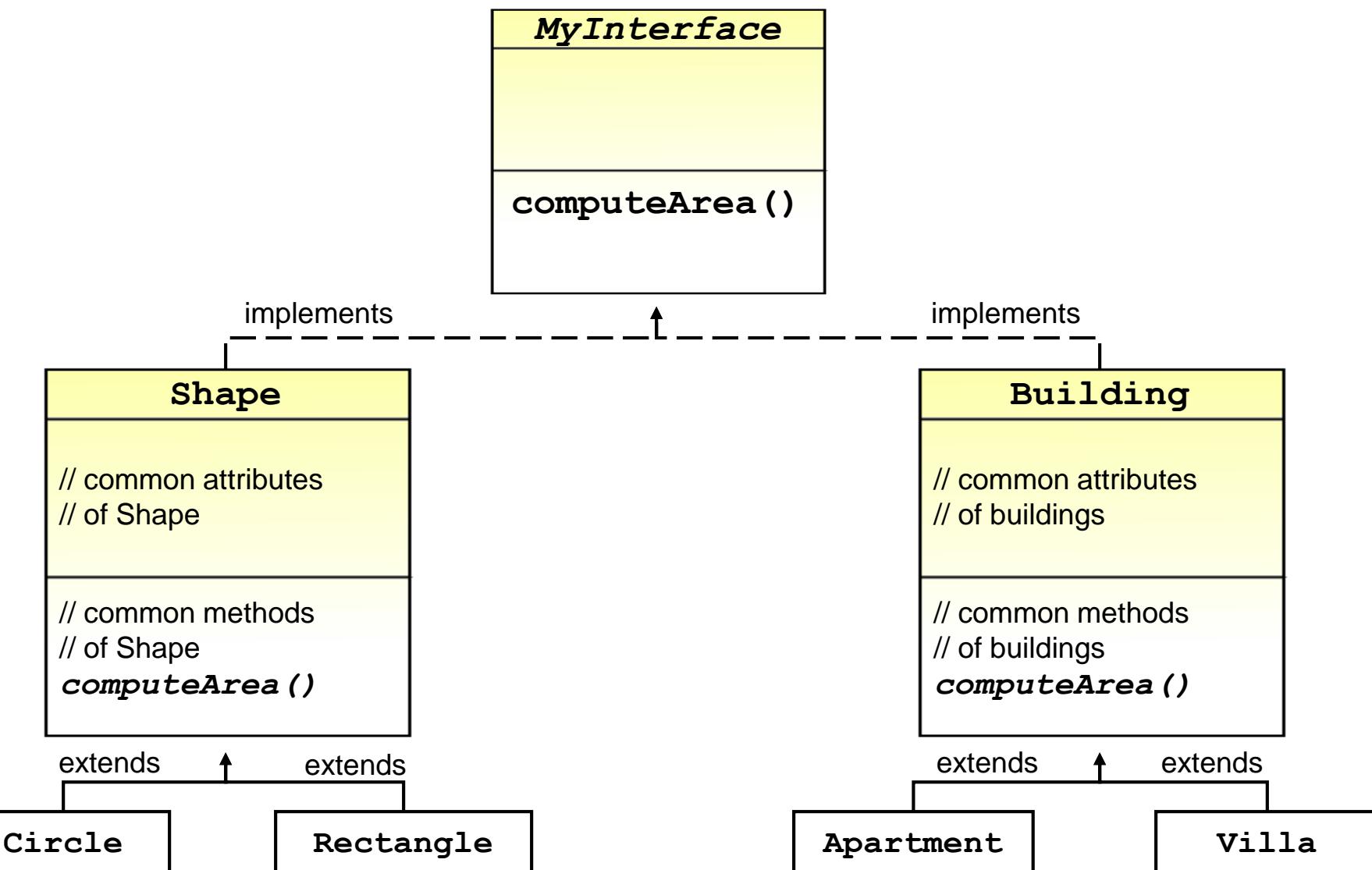
# Lesson 7

## 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**.

- 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.

# Example of Interfaces



Here is a simplified general form of a traditional interface:

```
Access specifier  interface
name
{
ret-type method-name1 (param-
list);
ret-type method-name2 (param-
list);
type var1 = value;
type var2 = 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**.

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

```
public interface Numbers
{
    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 Specifier class classname extends superclass implements interface {  
    // class-body  
}
```

# Implementing Interfaces

```
// Implement Numbers.
class ByTwos implements Numbers
{
    int start;
    int val;
    public 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 **Numbers** interface
- Notice that the methods `getNext()`, `reset()`, and `setStart()` are declared using the public access specifier

# Implementing Interfaces

```
public class Demo {  
    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("\n Resetting");  
        ob.reset();  
        for (int i = 0; i < 5; i++)  
            System.out.println("Next value is " + ob.getNext());  
        System.out.println("\n Starting at 100");  
        ob.setStart(100);  
        for (int i = 0; i < 5; i++)  
            System.out.println("Next value is " + ob.getNext());  
    } }
```

# Implementing Interfaces

```
// Implement Numbers.
class ByThrees implements
Numbers
{
    int start;
    int val;
    public 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 **Numbers** interface
- Notice that the methods `getNext()`, `reset()`, and `setStart()` are declared using the `public` access specifier

```
class Demo2
{
    public static void main (String args[])
    {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();
        Numbers 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());
        }
    }
}
```

- 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.

# Lesson 8

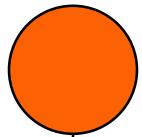
## Multi-Threading

- A Thread is
  - A single sequential execution path in a program
  - Used when we need to execute two or more program segments concurrently (multithreading).
  - 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.

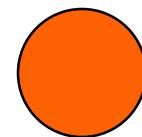
- Multiple threads do not mean that they execute in parallel when you're working in a single CPU.
  - Some kind of scheduling algorithm is used to manage the threads (e.g. Round Robin).
  - The scheduling algorithm is JVM specific (i.e. depending on the scheduling algorithm of the underlying operating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

- several thread objects that are executing concurrently:

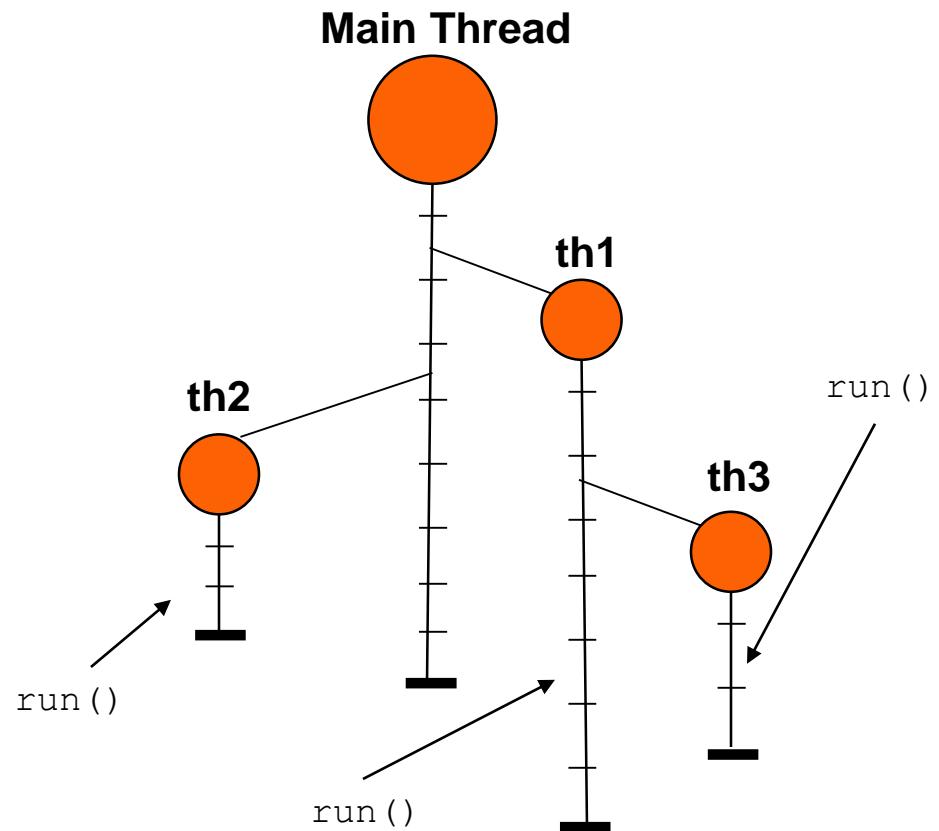
Garbage Collection Thread



Event Dispatcher Thread



Daemon Threads (System)



User Created Threads

- 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.

- **Class Thread**

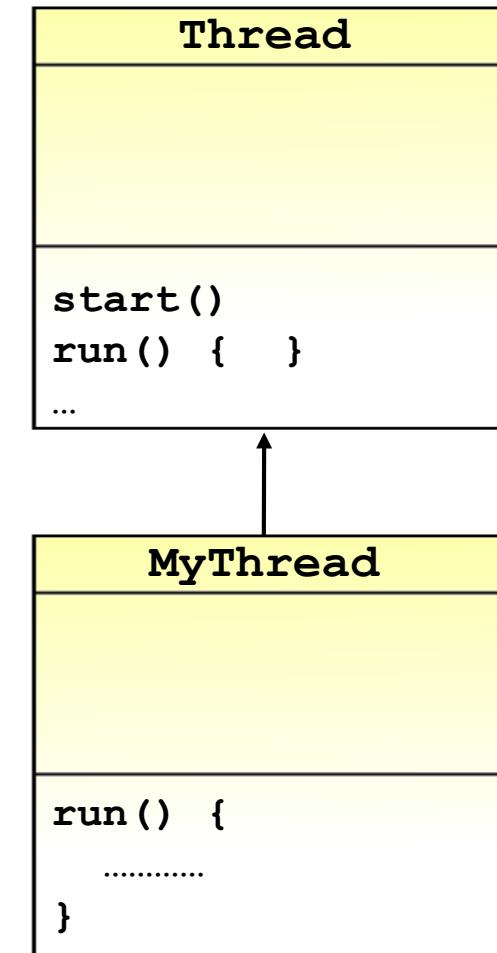
- **start()**
- **run()**
- **sleep()**
- **suspend() \***
- **resume() \***
- **stop() \***

- **Class Object**

- **wait()**
- **notify()**
- **notifyAll()**

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

- 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  
{  
    public void run() {  
        ... // write the job here  
    }  
}
```

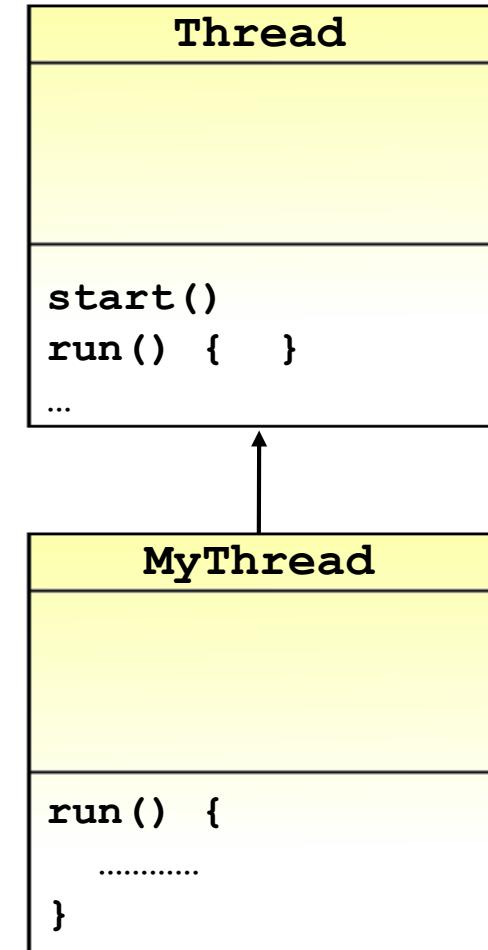
1

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

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

3.a

3.b



- 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  
{  
    public void run() {  
        ... // write the job here  
    }  
}
```

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

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

1

2

3

a

b

c

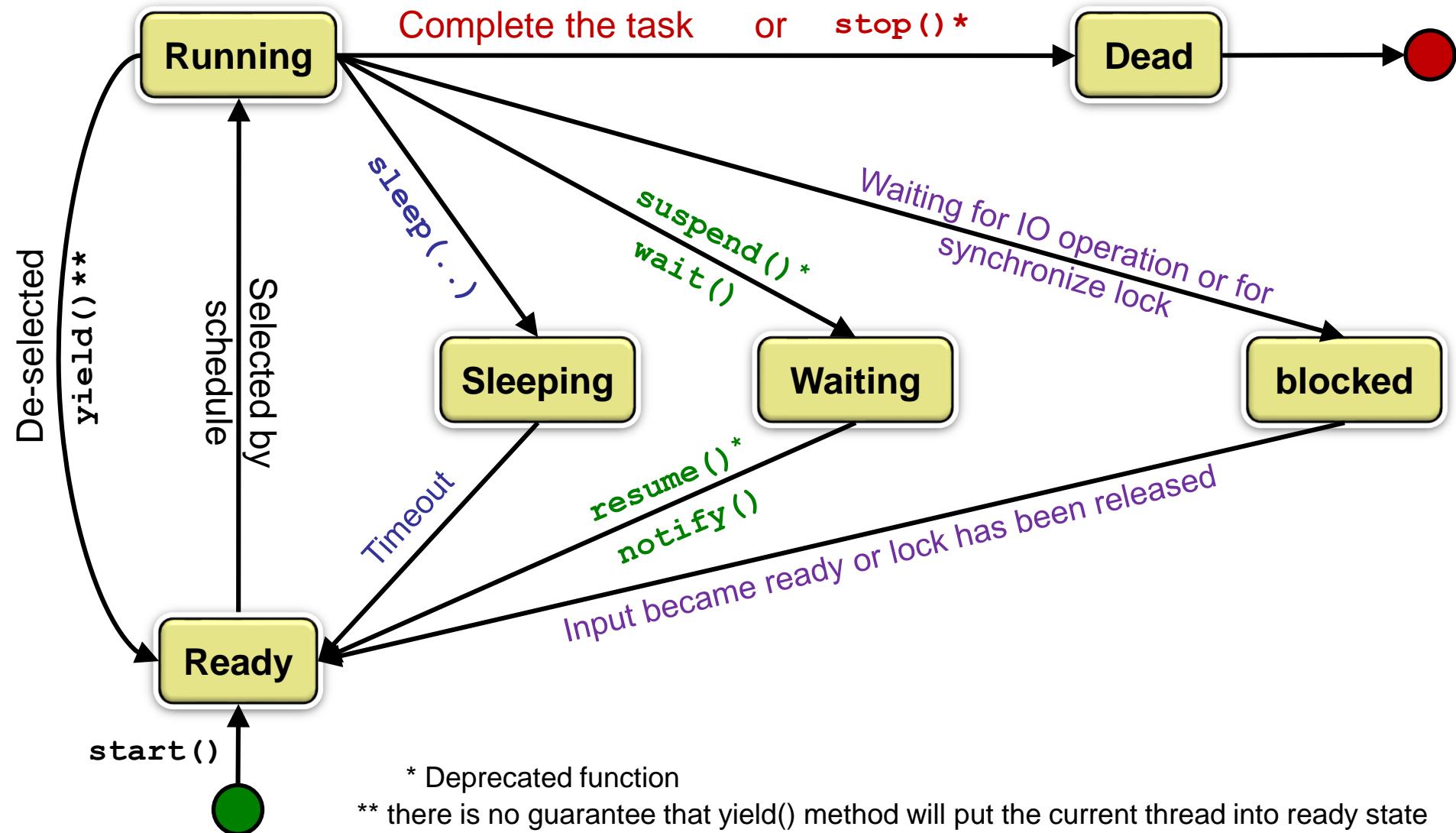
**Runnable****void run();****MyTask****void run(){  
.....  
}****Thread****Thread()  
Thread(Runnable r)  
start()  
run() { }**

- 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

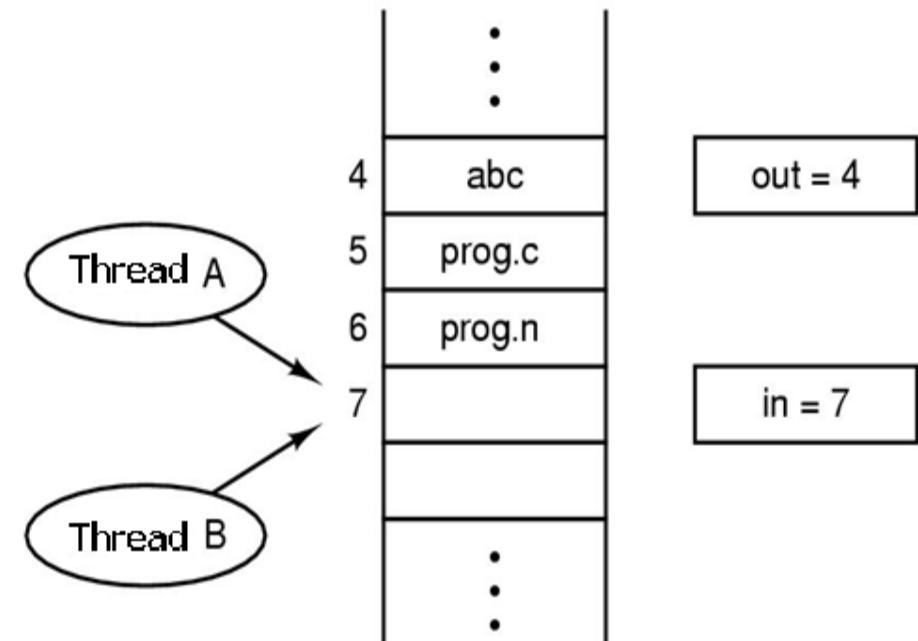
```
public class DateTimeApp extends Applet implements Runnable{  
    Thread th;  
    public void init(){  
        th = new Thread(this);  
        th.start();  
    }  
    public void paint(Graphics g){  
        Date d = new Date();  
        g.drawString(d.toString(), 50, 100);  
    }  
    public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

# Thread Life Cycle



- 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 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 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

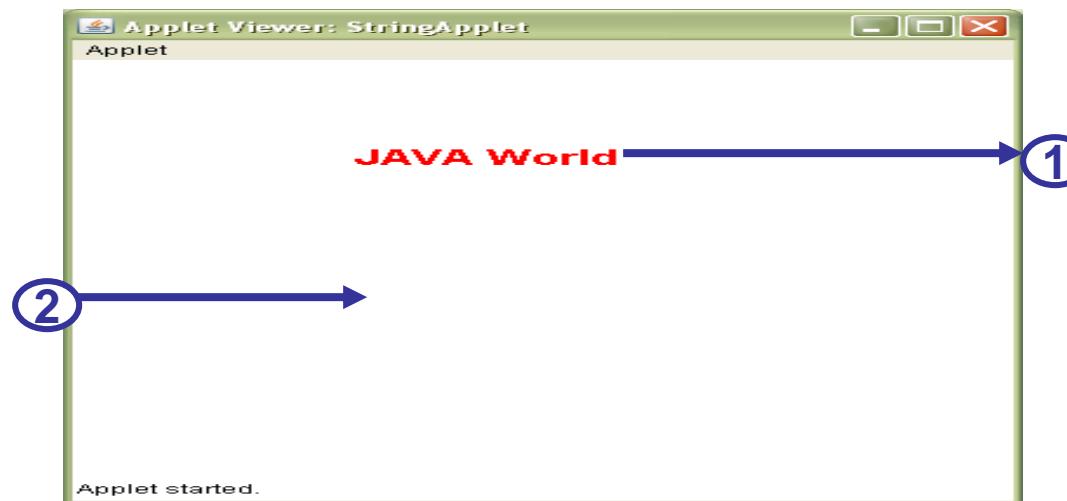
# 1. Date and Time Applet

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



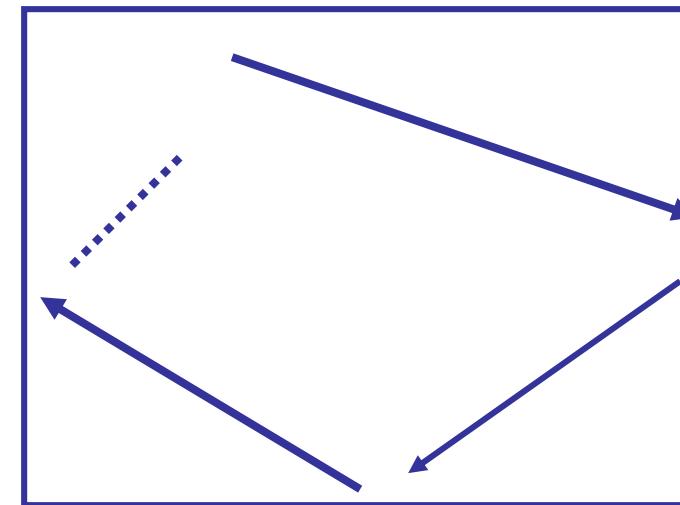
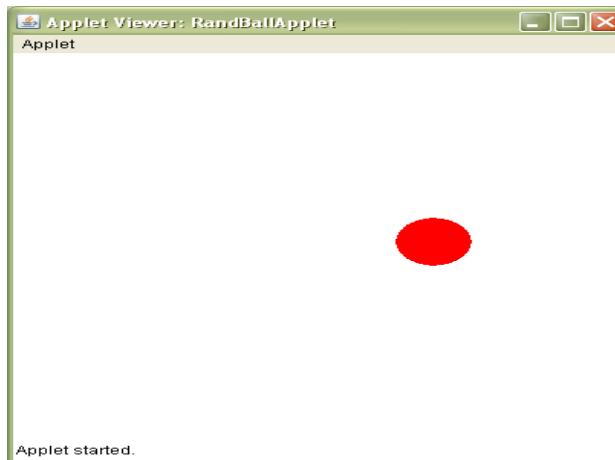
## 2. Text Banner Applet

- 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 9

## Inner Classes

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

- 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.

- 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.

- 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 OuterClass{
    private int x ;
    public void myMethod() {
        MyInnerClass m = new MyInnerClass();
        ...
    }

    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();  
    }  
    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 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;  
    public static class InnerClass{  
        public void aMethod(){  
            y = 10;          // OK  
            x = 33;         // wrong  
        }  
    }  
}
```

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

### 3. Local Inner Class

```
public class MyClass {  
    private int x ;  
    public void myMethod(final String str, final int a){  
        final int b = 5;  
        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();
    }
}
```

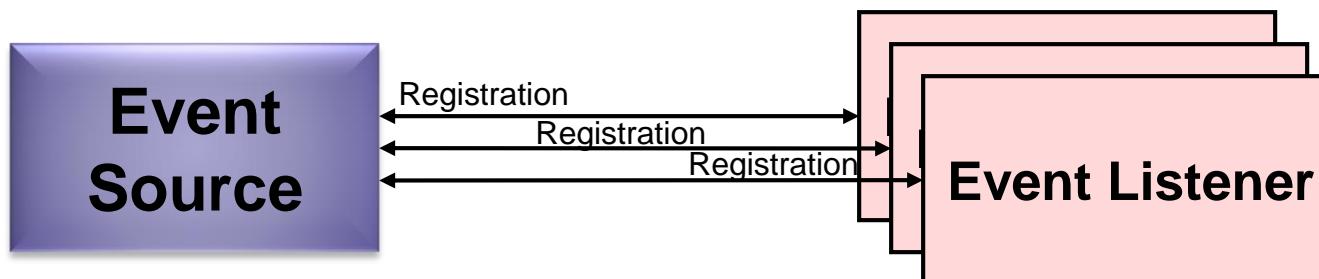
## 4. Anonymous Inner Class

- 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 10

## 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.



- 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.

- 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
```

- 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.

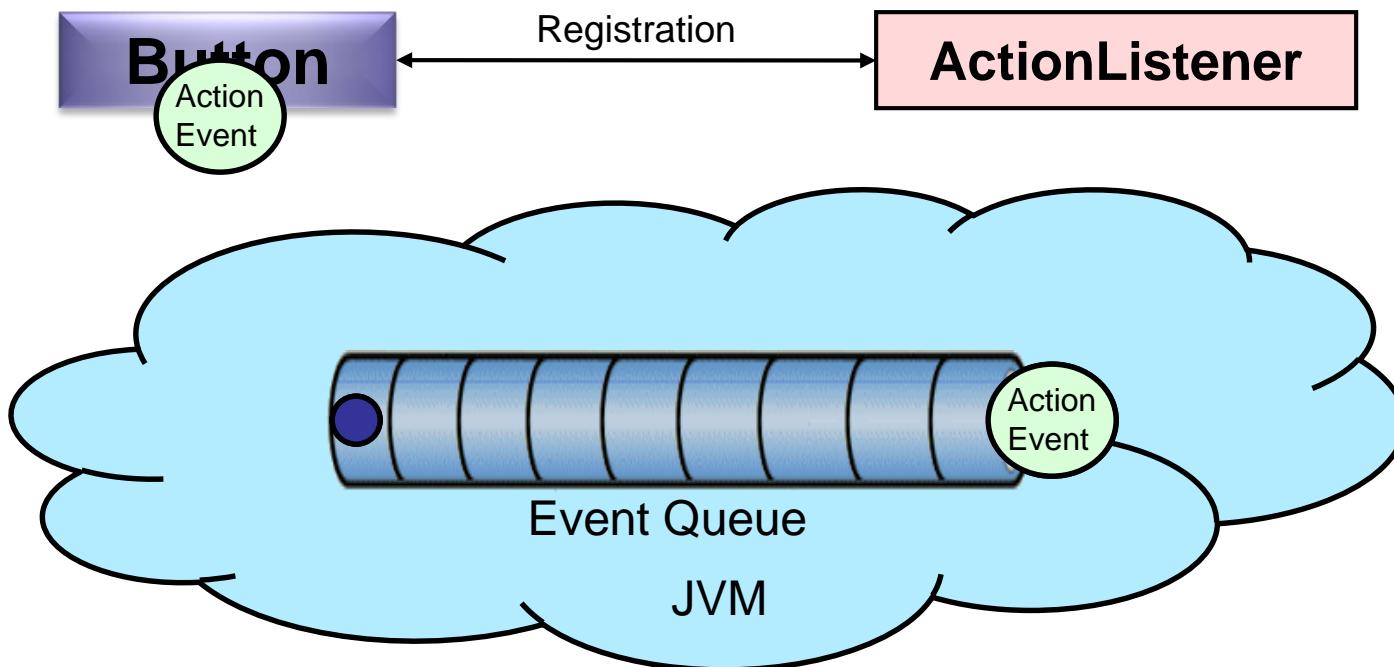
## 1. Write the listener code:

```
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



- 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.

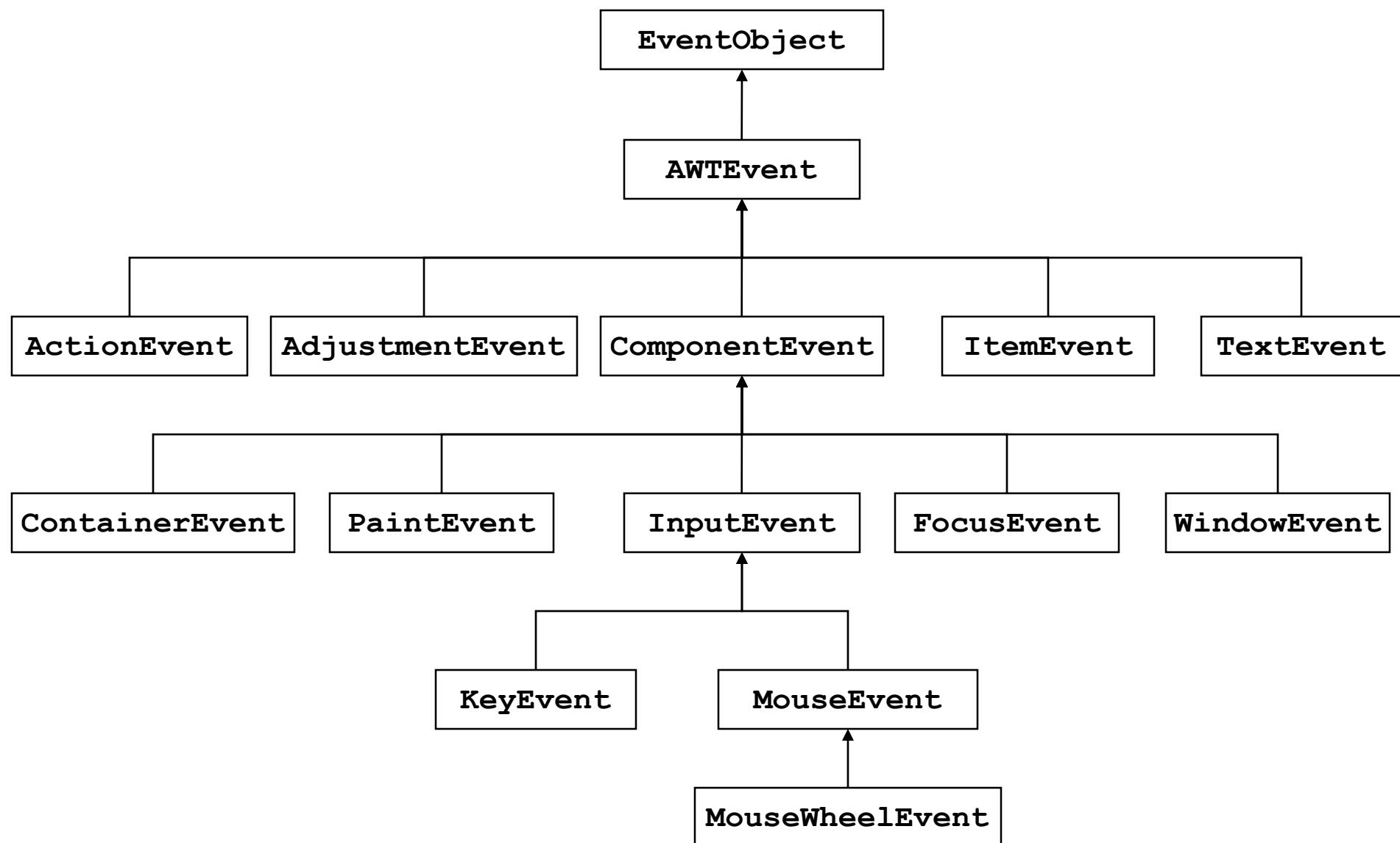
# Event Handling Example

```
public class ButtonApplet extends Applet{
    int x;
    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;
    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 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 (ComponentEvent e) componentRemoved (ComponentEvent e)

# Events Classes and Listener Interfaces

Event	Listener Interface(s)	Method(s)
<b>FocusEvent</b>	<b>FocusListener</b>	<b>focusGained (FocusEvent e)</b> <b>focusLost (FocusEvent e)</b>
<b>WindowEvent</b>	<b>WindowListener</b>	<b>windowClosed (WindowEvent e)</b> <b>windowClosing (WindowEvent e)</b> <b>windowOpened (WindowEvent e)</b> <b>windowActivated (WindowEvent e)</b> <b>windowDeactivated (WindowEvent e)</b> <b>windowIconified (WindowEvent e)</b> <b>windowDeiconified (WindowEvent e)</b>

# Events Classes and Listener Interfaces

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

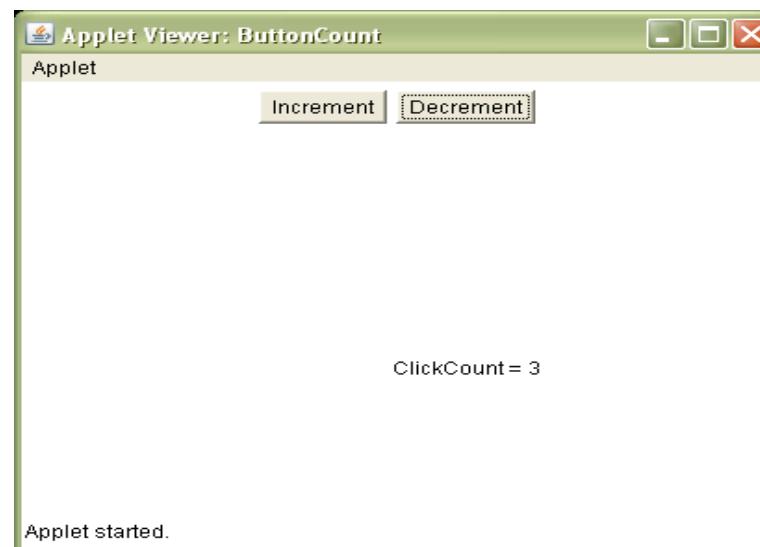
- When working with listener interfaces that have more than one method,
  - it is a tedious task to override all the methods of the interface when we only need to implement one of them.
- Therefore, for each listener interface with multiple methods,
  - there is a special **Adapter** class that has implemented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

- 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.

# Lab Exercise

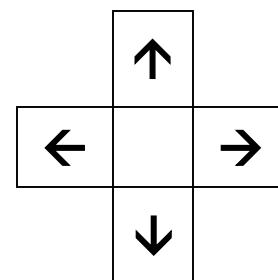
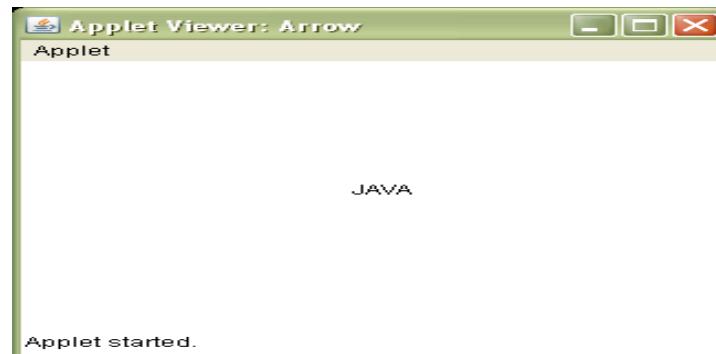
# 1. Button Count Applet

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



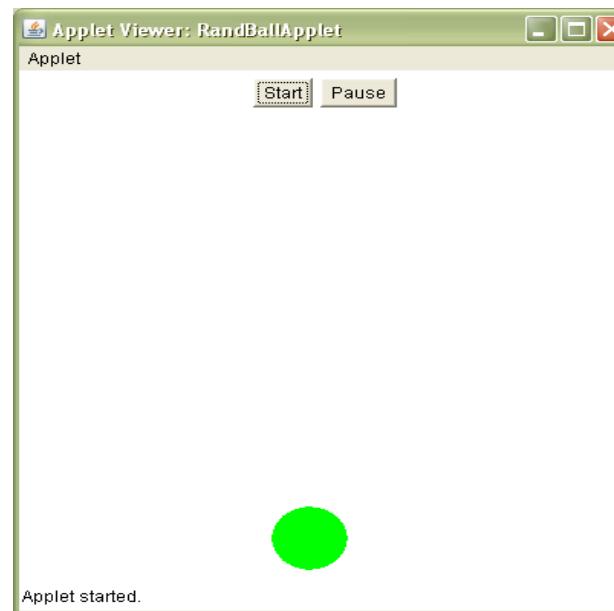
## 2. Moving Text Using Keyboard

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



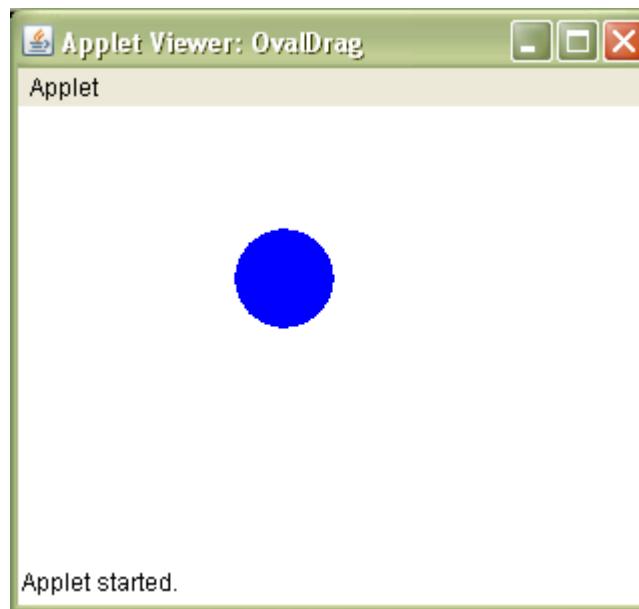
### 3. Play and Pause Animation

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



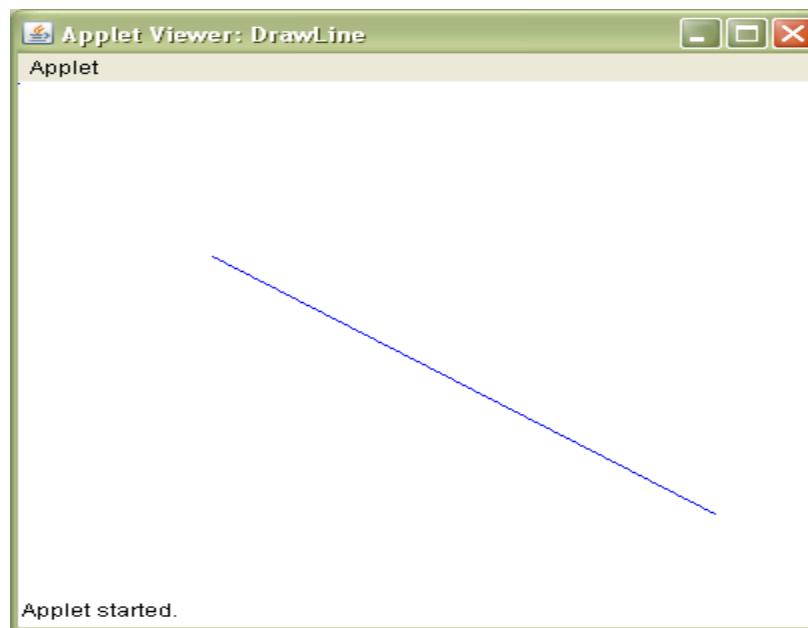
## 4. Drag Ball Applet

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



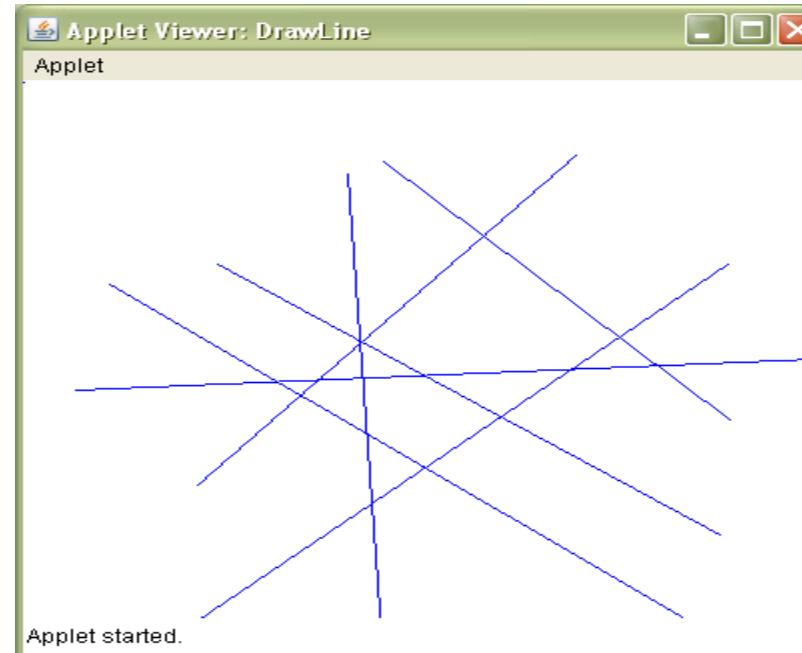
## 5. Draw Single Line Applet

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



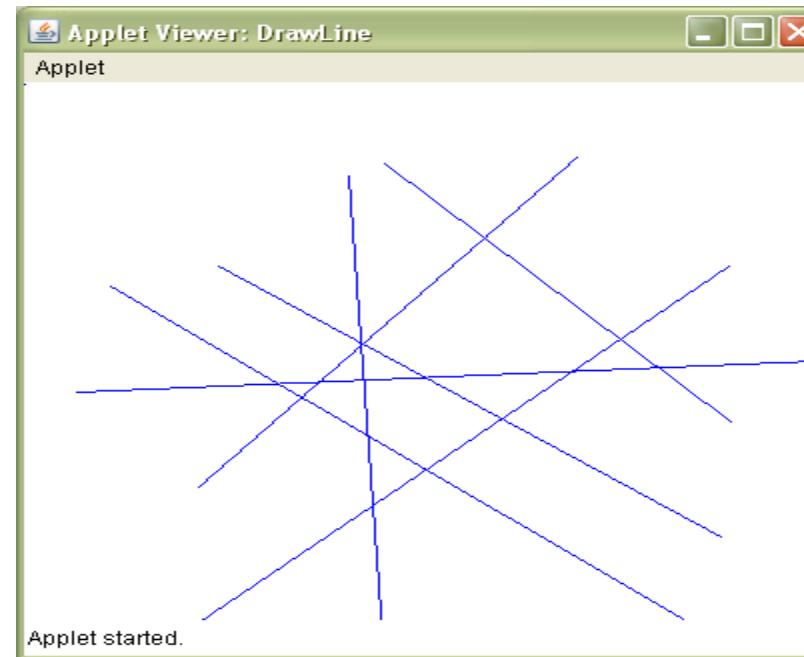
## 6. Draw Multiple Lines Applet

- 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. Draw Multiple Lines – using Vector

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



# Lesson 11

## Introduction to JavaFX 8

## □ AWT Abstract Window Toolkit :

- is an API to develop GUI or window-based application in java.
- AWT components are **platform-dependent** [ components are displayed according to the view of operating system ].
- AWT is **heavyweight** [ its components uses the resources of system ].

## □ Swing:

- It is built on the top of AWT.
- Java Swing provides platform-independent and lightweight components.



## □ AWT VS. Swing [ Desktop Application]:

No.	Java AWT	Java Swing
1)	AWT components are <b>platform-dependent</b> .	Java swing components are <b>platform-independent</b> .
2)	AWT components are <b>heavyweight</b> .	Swing components are <b>lightweight</b> .
3)	AWT <b>doesn't support pluggable look and feel</b> .	Swing <b>supports pluggable look and feel</b> .
4)	AWT provides <b>less components</b> than Swing.	Swing provides <b>more powerful components</b> such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT <b>doesn't follows MVC</b> (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing <b>follows MVC</b> .

- **F3 (Form Follows Function)** by **Chris Oliver**.
- **F3** is a declarative scripting language with good support of IDE, and compile time error reporting unlike javascript.
- **Chris Oliver** became a **Sun** employee through **Sun** acquisition of **See Beyond Technology Corporation** in September 2005.
- At JavaOne 2007 , Its name was changed to **JavaFX** [Open Source].
- The first version of **JavaFX** Script was an interpreted language, and was considered a prototype of the compiled JavaFX Script language.

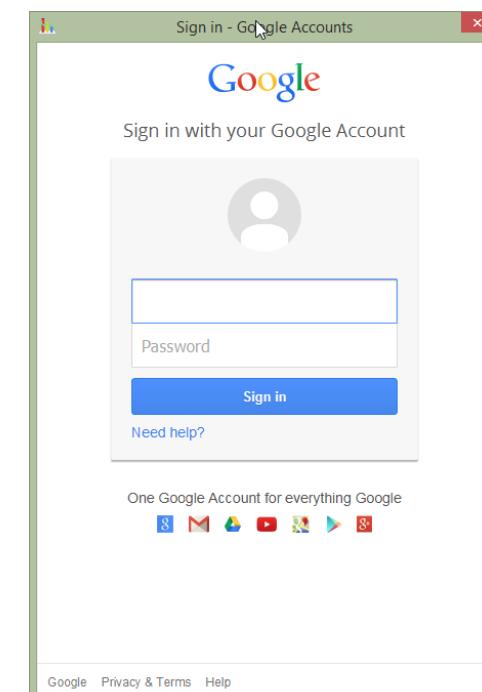
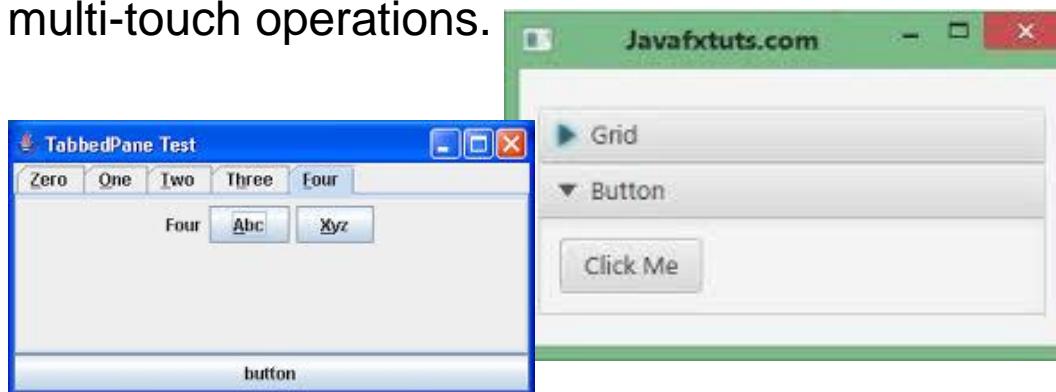
- At JavaOne 2009, the JavaFX SDK 1.2 was released.
- At JavaOne 2010, the JavaFX SDK 2.0 was announced.
- **JavaFX 2.0** road-map :
  - Deprecating JavaFX script.
  - Porting all JavaFX scripting features into JavaFX 2.0 APIs.
  - Providing web component for embedding HTML and JavaScript into JavaFX code.
  - Enable JavaFX interoperability with swing.
- At JavaOne 2011, the JavaFX SDK 2.0 was released

- **JavaFX** is a next generation graphical user interface toolkit.
- It is intended to replace java **Swing** as the standard GUI library for **JavaSE**.
- is a set of graphics and media packages that enables developers to **design**, **create**, **test**, **debug**, and **deploy** rich client applications that operate consistently across diverse platforms.
- **JavaFX** has included a feature of customized style using Cascading Style Sheets (**CSS**) style.

- It is easy to learn because it is very similar to Java **swing**.
- **Swing** application can be updated with JavaFX features
- A new language is added to JavaFX called **FXML**, which is XML that is used only to define the interface of an application, So that it completely separated from the logic of the code.
- The library of JavaFX is created using Java native code or Java API.
- JavaFX is also platform independent so that it can run any platform using the JVM.

## □ JavaFX supports:

- **Webview:** A web component can be embedded inside the JavaFX application to view web pages.
- many extra features like date-picker, accordion pane, tabbed pane and pie-chart.
- animations, 2D and 3D graphics .
- powerful way of design using CSS.
- multi-touch operations.



# Java FX Components



## □ AWT

- Platform Specific.

## □ Swing

- Only for Desktop Applications

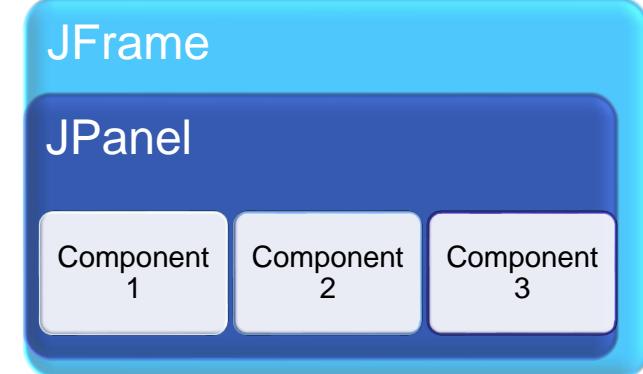


## □ JavaFX

- Desktop, websites, Handheld Devices friendly.

- JavaFX has several interesting controls that Swing doesn't have, such as the collapsible TitledPane control and the Accordion control.
- The `javafx.scene.effect` package contains a number of classes that can easily apply special effects to any node in the scene graph.
- JavaFX has built-in support for sophisticated animations that can be applied to any node in the scene graph.

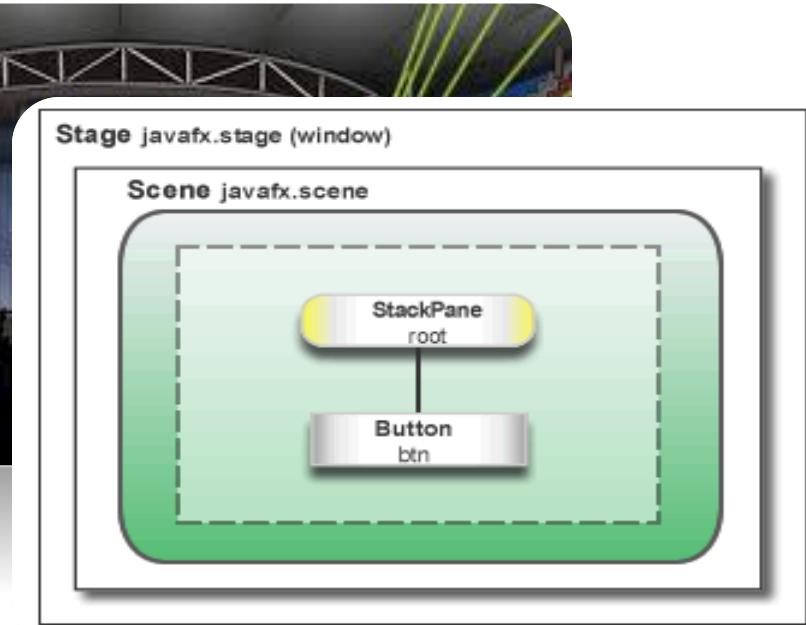
## In Swing,



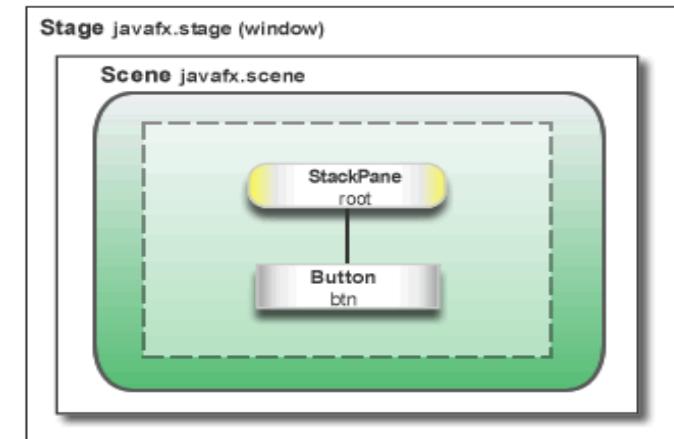
- The class that holds your user interface components is called a **JFrame** class.
- A **JFrame** is an empty window to which you can add a **JPanel**, which serves as a container for your user-interface elements.
- A Swing application is actually a class that extends the JFrame class. To display user-interface components, you add components to a JPanel and then add the panel to the frame.

# Java Swing Vs JavaFX

In JavaFX, All the world is Stage



In JavaFX, All the world is Stage

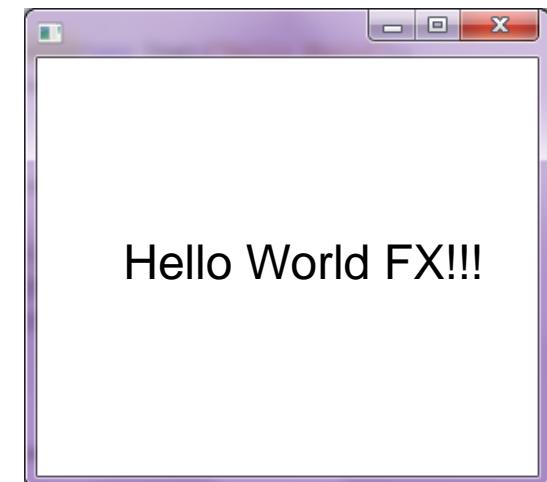


- A **stage** is the highest level container .
- The individual controls and other components that make up the user interface are contained in a **scene** .
- An application can have more than one scene, but only one of the scenes can be displayed on the stage at any given time.
- A scene contains a **scene graph**, is a collection of all the elements [**nodes**] that make up a user interface .

- There are three ways to create a hello world program
  1. Simple code using classes
  2. Using FXML (XML)
  3. Using Scene builder

# Hello world program in JavaFX

```
public class HelloWorld extends Application{  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
  
        Text helloworld = new Text("Hello World FX!!!");  
  
        StackPane rootPane = new StackPane(helloworld);  
  
        Scene scene = new Scene(rootPane, 400, 300);  
  
        primaryStage.setScene(scene);  
        primaryStage.show();  
  
    }  
  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
}
```



## Application Class

- The entry point for JavaFX applications.
- The Application class is an abstract class with a single abstract method **start**.
- The application class has three important methods:

Method	Signature
init	<code>public void init() throws Exception</code>
start	<code>public abstract void start(Stage primaryStage) throws Exception</code>
stop	<code>public void stop() throws Exception</code>

## Application Class

### launch() method:

- This method is typically called from the main method().
- It **must not** be called more than once or an exception will be thrown.
- The launch method does not return until the application has exited , either via a call to **Platform.exit** or **all of the application's windows have been closed** and the Platform attribute **implicitExit** is set to true.

```
public static void launch(java.lang.Class<? extends Application> appClass,  
                        java.lang.String... args)  
  
public static void launch(java.lang.String... args)
```

# The Life Cycle of a JavaFX Application

The launch() method waits for the JavaFX application to finish

JavaFX Runtime

Creates JavaFX Application Object in Application Thread

JavaFX Launcher Thread

Call init() on the created Object

JavaFX Application Thread

-  
calls the start(Stage stage) method of the specified Application class.

Not allowed to create a stage or a scene

When the application finishes, the JavaFX Application Thread calls the stop() method of the specified Application

- ❑ JavaFX runtime is responsible for creating several threads
- ❑ At different phases in the application, threads are used to perform different tasks.
- ❑ The JavaFX runtime creates, among other threads, two threads:
  - ❑ JavaFX-Launcher Thread
  - ❑ JavaFX Application Thread
- ❑ The launch() method of the Application class create these threads.

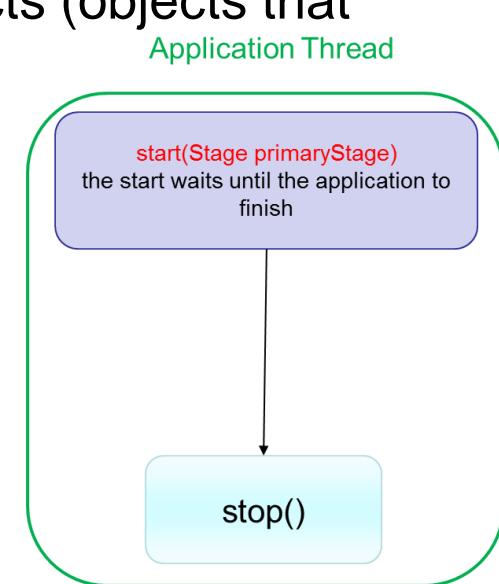
- ❑ During the lifetime of a JavaFX application, the JavaFX runtime calls the following methods of the specified JavaFX Application class in order:
  - ❑ The no-args constructor [ in **Application** Thread]
  - ❑ The init() method [ in **Launcher** Thread]
  - ❑ The start() method [ in **Application** Thread]
  - ❑ The stop() method [ in **Application** Thread]

## Launcher Thread

- ❑ Is the thread that is used for launching the application.
- ❑ Constructing and modifying the **Stage** on the **launcher thread** **is not allowed** as it will throw an exception.
- ❑ Also modifying objects that are attached to the scene graph.

## Application Thread

- ❑ Is the thread that is used to invoke the `start()` and `stop()` methods.
- ❑ This thread is used to construct and modify the JavaFX **Stage** and **Scene**, Process input events, running **animation** timeline, and apply modifications to live objects (objects that are attached to the scene).



- The following example code illustrates the life cycle of a JavaFX application.

```
public class LifeCycleTest extends Application {  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    public LifeCycleTest() {  
        String name = Thread.currentThread().getName();  
        System.out.println("Constructor() method: current Thread:" + name);  
  
    }  
  
    @Override  
    public void init() throws Exception {  
        String name = Thread.currentThread().getName();  
        System.out.println("init() method: current Thread:" + name);  
        super.init();  
    }  
}
```

# The Life Cycle of a JavaFX Application

```
public void start(Stage primaryStage) {  
  
    String name = Thread.currentThread().getName();  
    System.out.println("start() method: current Thread:" + name);  
  
    StackPane root = new StackPane();  
    root.getChildren().add(new Text("Hello Life Cycle"));  
    Scene scene = new Scene(root, 300, 250);  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}  
  
@Override  
public void stop() throws Exception {  
    String name = Thread.currentThread().getName();  
    System.out.println("Stop() method: current Thread:" + name);  
    super.stop();  
}  
}
```

Constructor() method: current Thread:JavaFX Application Thread  
init() method: current Thread:JavaFX-Launcher  
start() method: current Thread:JavaFX Application Thread  
Stop() method: current Thread:JavaFX Application Thread

- ❑ Is the JavaFX application Platform support class.
- ❑ It can check the current running thread (application thread or not),
- ❑ enqueue a task into the application thread,
- ❑ and control the default exit behavior.

- public static boolean **isFxApplicationThread** () //Returns true if the calling thread is the JavaFX Application Thread
- public static void **runLater** (Runnable runnable) //Run the specified Runnable on the JavaFX Application Thread in the future
- public static void **exit** () //Causes the JavaFX application to terminate

# Program Structure

A JavaFX application consists of three main components.

Nodes.

Scene.

Stage.



The **Node** is the main actor of the application, and the visible component in our application.

**Scene** is the component where the nodes are displayed on it.

**Stage** is the base for the scene, and nodes.

# Node

A **scene graph** is a set of tree data structures where every item is a Node.

Each item is either a "**leaf**" with zero sub-items or a "**branch**" with zero or more sub-items.

A node may occur at most **once** anywhere in the scene graph.

Node objects may be constructed and modified on any thread as long they are not yet attached to a Scene.

Modifying nodes that are already attached to a Scene (**live objects**), on the **JavaFX Application Thread** only.

One of the greatest advantages of JavaFX is the ability to use CSS to style your nodes in the scene graph.

JavaFX CSS are based on the W3C CSS version 2.1 specification.

The default style sheet for JavaFX applications is *caspian.css*, which is found in the JavaFX runtime JAR file, jfxrt.jar. This style sheet defines styles for the root node and the UI controls.

To change the default style of a node you can use the **setStyle** method.

```
helloworld.setStyle("-fx-fill: #09f415;"  
                    + "-fx-cursor: hand;");
```

The JavaFX **Scene** class is the container for all content in a **scene graph**.

The application must specify the **root Node** for the scene graph by setting the root property.

```
StackPane rootPane = new StackPane(helloWorld);  
Scene scene = new Scene(rootPane, 400, 300);
```

The scene's size may be initialized by the application during construction. If no size is specified, the scene will automatically compute its initial size based on the preferred size of its content.

Scene objects must be constructed and modified on the **JavaFX Application Thread**.

# Scene and Style

To create a style class we first create a .css file to indicate a CSS style sheet.

```
.root{  
    -fx-font: 25px "sans-serif";  
    -fx-fill: #eb2020;  
}
```

To apply this style to our scene we must add this sheet to the scene styles.

```
scene.getStylesheets().add(getClass()  
    .getResource("styles/styles.css").toString());
```

# Scene and Style

To add a certain class to a node you can use the `getStyleClasses().add()` method.

MyStyles.css

```
.myStyleClass{  
    -fx-fill: #115ee5;  
    -fx-font: 25px sans-serif;  
}
```

Application `start()` method

```
@Override  
public void start(Stage primaryStage) {  
  
    Text txt = new Text("Hello World");  
  
    StackPane root = new StackPane();  
    root.getChildren().add(txt);  
  
    Scene scene = new Scene(root, 300, 250);  
  
    scene.getStylesheets().add(getClass()  
        .getResource("../styles/MyStyles.css").toString());  
    txt.getStyleClass().add("myStyleClass");  
  
    primaryStage.setTitle("Hello World!");  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}
```

# Scene and Style

Any node that will be displayed on the screen must be attached to the scene somehow.

Each node can have an **Id** property to identify it.

```
helloworld.setId("text");
```

The node can be later re-located using the lookup method.

This is very helpful when using CSS styles, as we will not write a style for each node one by one. But we use style classes.

```
Node x= scene.lookup("text");
```

# Scene and Style

You can create a style class for a certain node in the scene using the hash symbol (#) and the node Id.

MyStyle.css

```
#text{  
    -fx-font: 25px "sans-serif";  
    -fx-fill: #eb2020;  
}
```

Application **start()** method

```
@Override  
public void start(Stage primaryStage) {  
  
    Text txt = new Text("Hello World");  
    txt.setId("text");  
    StackPane root = new StackPane();  
    root.getChildren().add(txt);  
  
    Scene scene = new Scene(root, 300, 250);  
  
    scene.getStylesheets().add(getClass()  
        .getResource("../styles/MyStyles.css").toString());  
  
    primaryStage.setTitle("Hello World!");  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}
```

# Stage

The JavaFX Stage class is the top level JavaFX container.

The primary Stage is constructed by the platform.

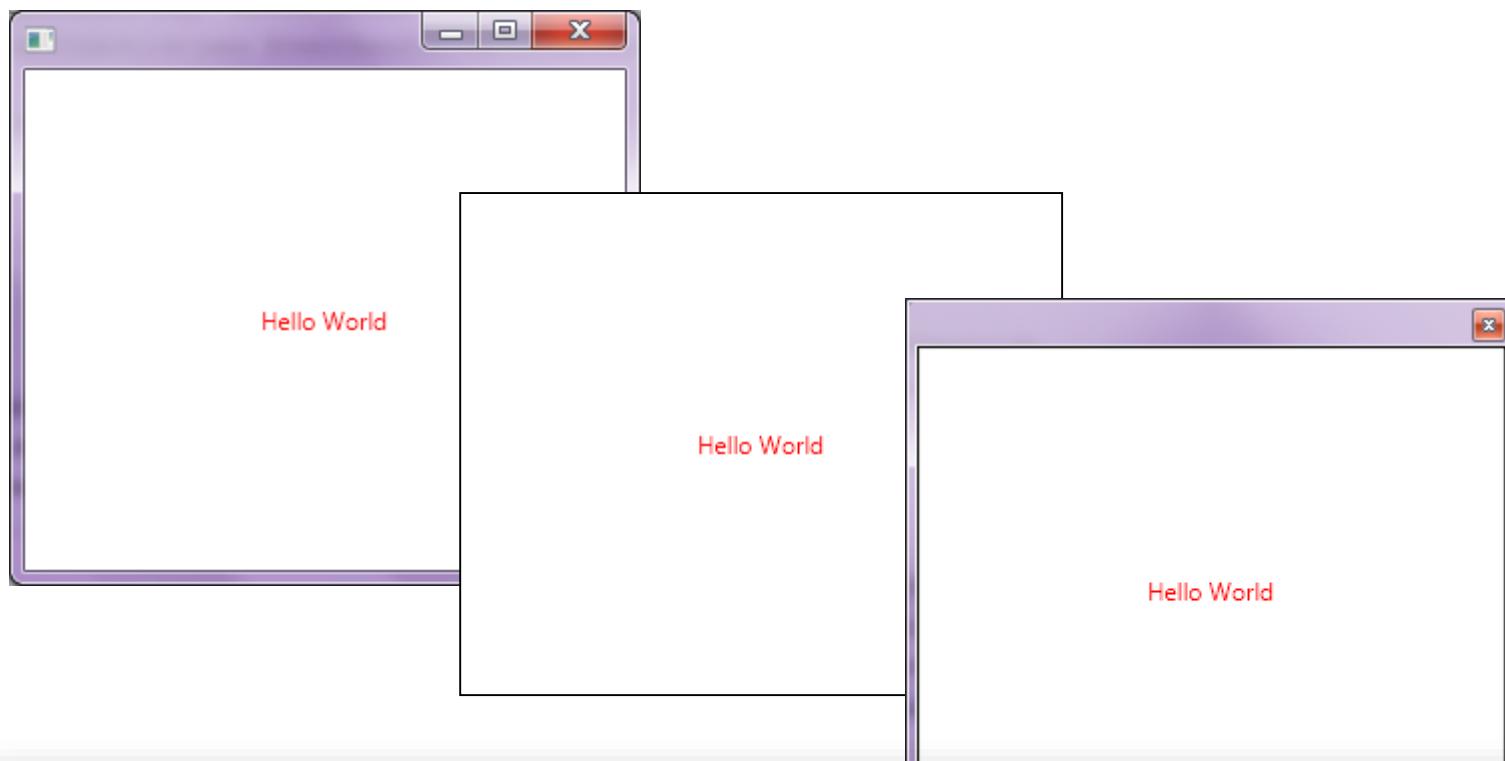
Stage object must be constructed and modified on the **JavaFX Application Thread**.

A stage has one of the following styles:

- **StageStyle.DECORATED** : a stage with a solid white background and platform decorations.
- **StageStyle.UNDECORATED** : a stage with a solid white background and no decorations.
- **StageStyle.TRANSPARENT** : a stage with a transparent background and no decorations.
- **StageStyle.UTILITY** : a stage with a solid white background and minimal platform decorations.
- **The style must be initialized before the stage is made visible.**

# Stage

A stage has one of the following styles:



```
primaryStage.initStyle(StageStyle.UTILITY);
```

A stage can optionally have an owner Window.

```
public final void initOwner(Window owner)
```

When a parent window is closed, all its descendant windows are closed.

A stage has one of the following modalities:

- **None**: stage that does not block any other window.
- **Window Modal**: a stage that blocks input events from being delivered to all windows from its owner (parent) to its root.
- **Application Modal**: a stage that blocks input events from being delivered to all windows from the same application, except for those from its child hierarchy.

# Lab Exercise

# Hello World

- Create the Hello World application to match the following style.
  1. Use JavaFX code [ [Reflection](#) , [LinearGradient](#) ]
  2. **Bonus:** Use CSS style sheet.



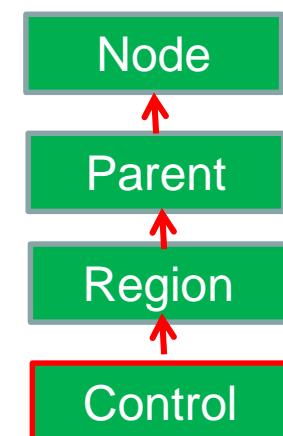
- <https://docs.oracle.com/javase/8/javafx/api/toc.htm>
- <https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>

# Building UI Using JavaFX

# Building UI Using JavaFX

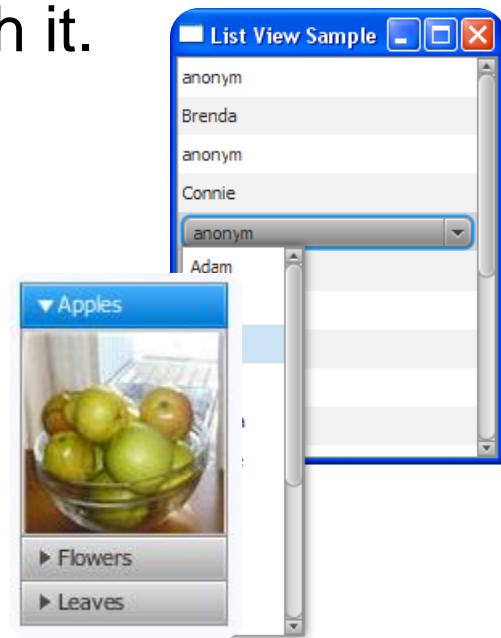
## Basic Controls

- Class **Control** is the base class for all javaFX Controls.
- Class **Control** is a sub-class of class **Node**, so it can be treated as node in the scene plus its variables and behaviours as control to support user interactions.
- controls support explicit skinning to make it easy to leverage the functionality of a control while customizing its appearance (Context menu, skin, Tooltip).



# Labeled controls

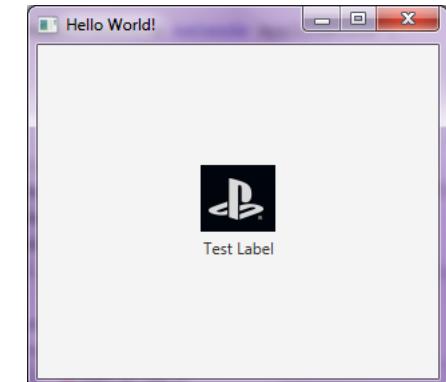
- A Labeled Control is one which has as part of its user interface a textual content associated with it.
- It has four sub-classes:
  - **Cell**: used for virtualized controls such as:
    - ListView, TreeView, and TableView.
  - **Label**: is a non-editable text control.
  - **TitledPane**: panel with a title that can be opened and closed.
  - **ButtonBase**: Base class for button-like UI Controls, including Hyperlinks, Buttons, ToggleButtons, CheckBoxes, and RadioButtons.



# Labeled controls

- We can customize a Labeled control to hold also images and text.

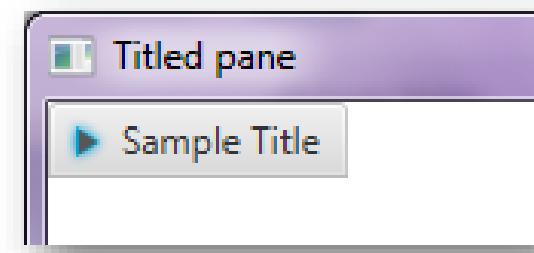
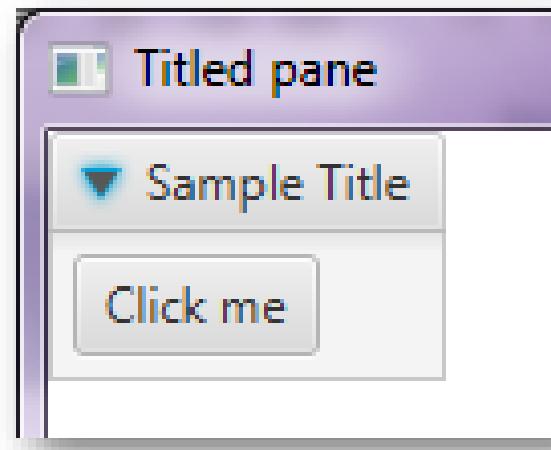
```
Image img = new Image(getClass()
    .getResourceAsStream("images/smile.png"));
ImageView view = new ImageView(img);
Label label = new Label("Test Label", view);
label.setContentDisplay(ContentDisplay.TOP);
```



- The panel in a **TitledPane** can be any Node such as UI controls or groups of nodes added to a layout container.
- **Note:** the inherited properties from class Labeled are used to manipulate the header area not the content area.
- It is **not recommended** to set the MinHeight, PrefHeight, or MaxHeight for this control. Unexpected behavior will occur because the TitledPane's height changes when it is opened or closed.

# Labeled controls

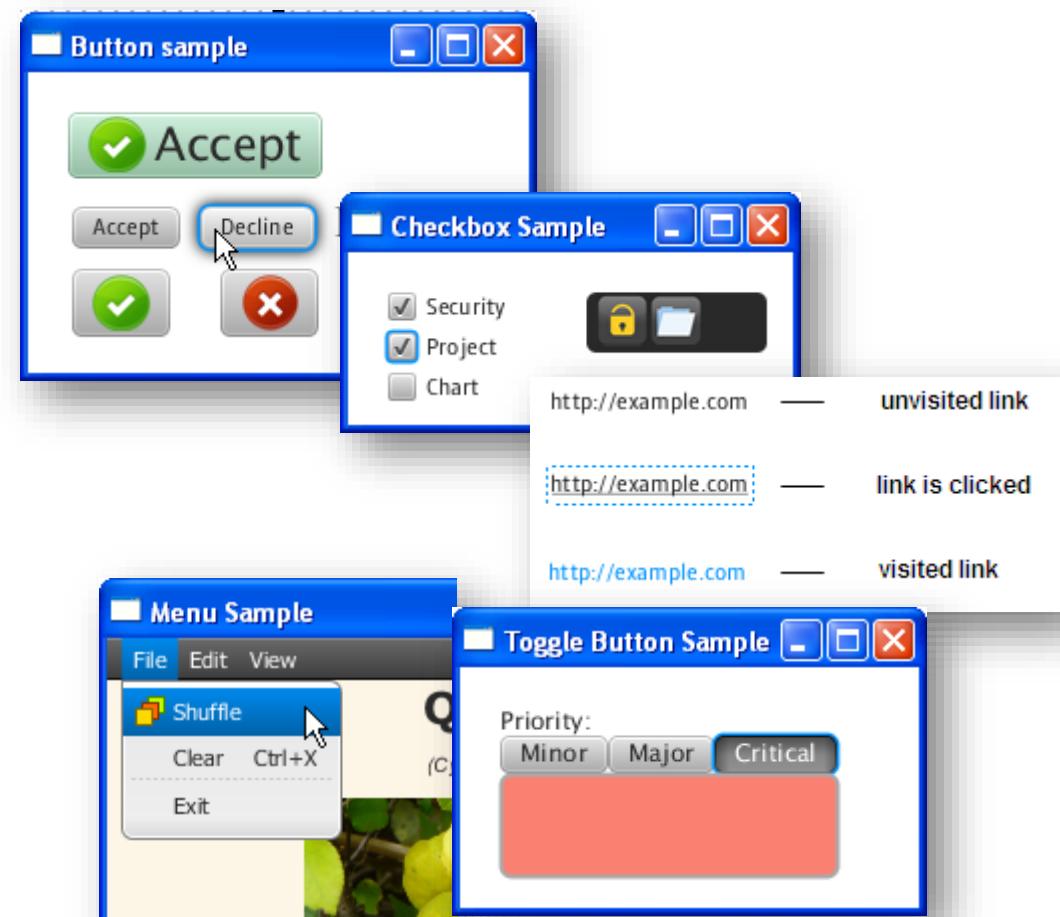
```
TitledPane pane = new TitledPane("Sample Title",new Button("Click me"));  
Scene scene = new Scene(new Group(), 300, 400);  
Group root = (Group) scene.getRoot();  
root.getChildren().add(pane);
```



# ButtonBase class

- The **ButtonBase** class is an extension of the **Labeled** class. It can display text, an image, or both.

- Sub-classes are:
  - Button.
  - CheckBox.
  - HyperLink.
  - MenuButton.
  - ToggleButton.



# Button

- A simple button control.
- The button control can contain text and/or a graphic.



- A button control has three different modes:
  - **Normal:** A normal push button.
  - **Default:** A default Button is the button that receives a keyboard **VK\_ENTER** press, if no other node in the scene consumes it.
  - **Cancel:** A Cancel Button is the button that receives a keyboard **VK\_ESC** press, if no other node in the scene consumes it.

# Button

```
@Override  
public void start(Stage primaryStage) throws Exception {  
    Button b1 = new Button("Normal");  
    Button b2 = new Button("Default"); ←  
    Button b3 = new Button("Cancel");  
  
    b2.setDefaultButton(true); ←  
    b3.setCancelButton(true);  
  
    FlowPane root = new FlowPane();  
    root.getChildren().addAll(b1,b2,b3);  
  
    Scene scene = new Scene(root, 300, 400);  
  
    primaryStage.setTitle("Button Example");  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}
```

Button Creation

Change Button Type



# CheckBox

- A **tri-state** selection Control typically skinned as a box with a checkmark or tick mark when checked.
- A **CheckBox** control can be in one of three states:

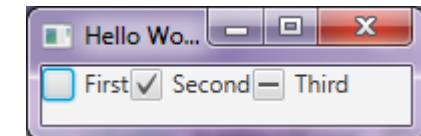
State	indeterminate	Checked
Checked	false	true
unChecked	false	false
undefined	true	--

- when the checkbox is undefined, it cannot be selected or deselected.

```
cb1.setIndeterminate(false);
cb1.setSelected(false);

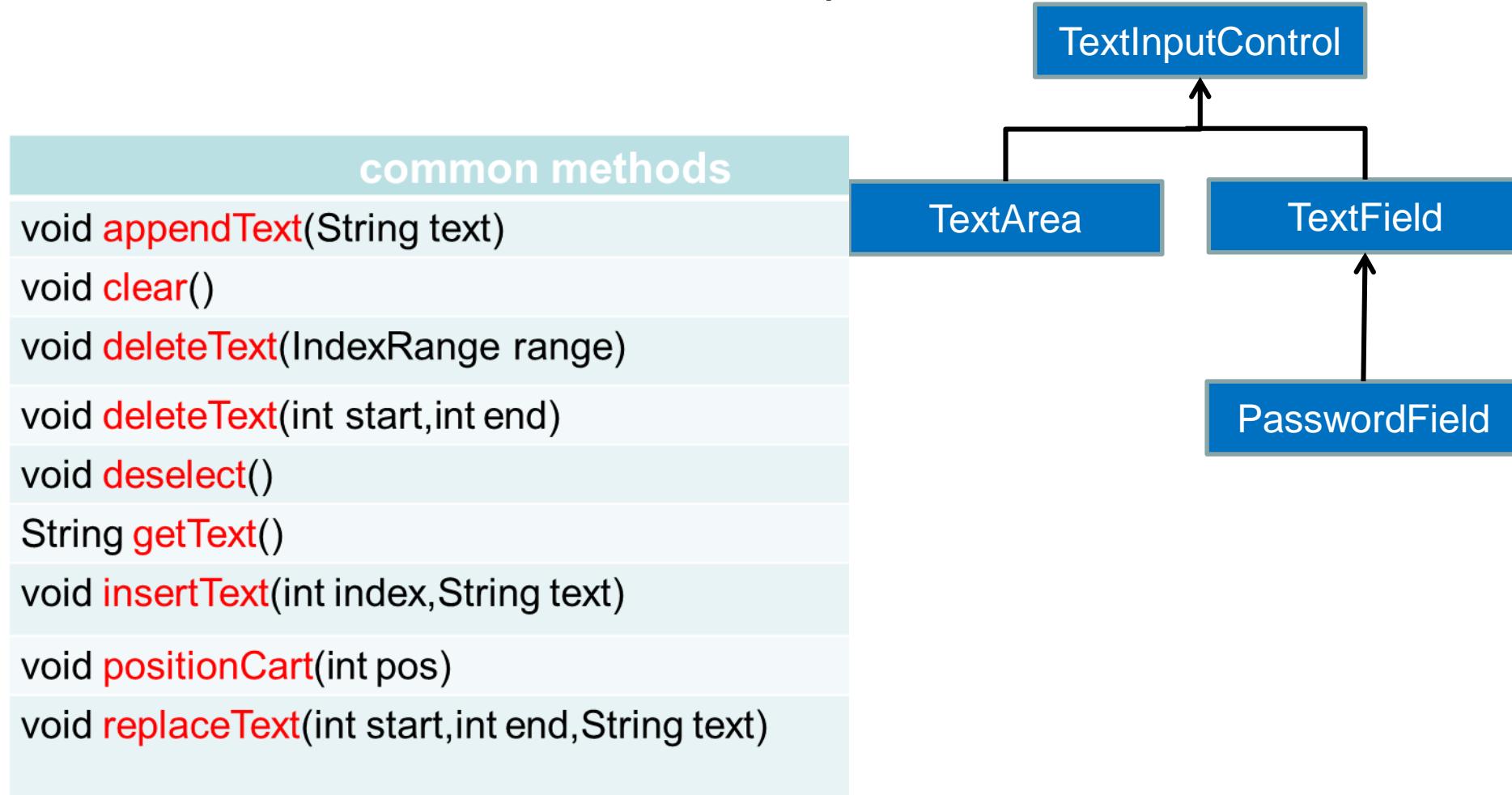
cb1.setIndeterminate(false);
cb2.setSelected(true);

cb3.setIndeterminate(true);
cb3.setSelected(false);
```



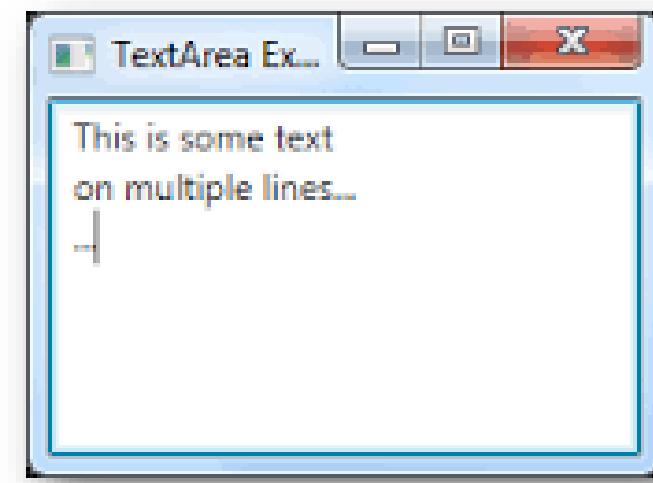
# TextInputControl

- Abstract base class for text input controls.



# TextArea

- Text input component that allows a user to enter multiple lines of plain text.
- you can use the `setPrefRowCount()`, and `setPrefColCount()` to adjust the preferred size of the TextArea.



# TextField

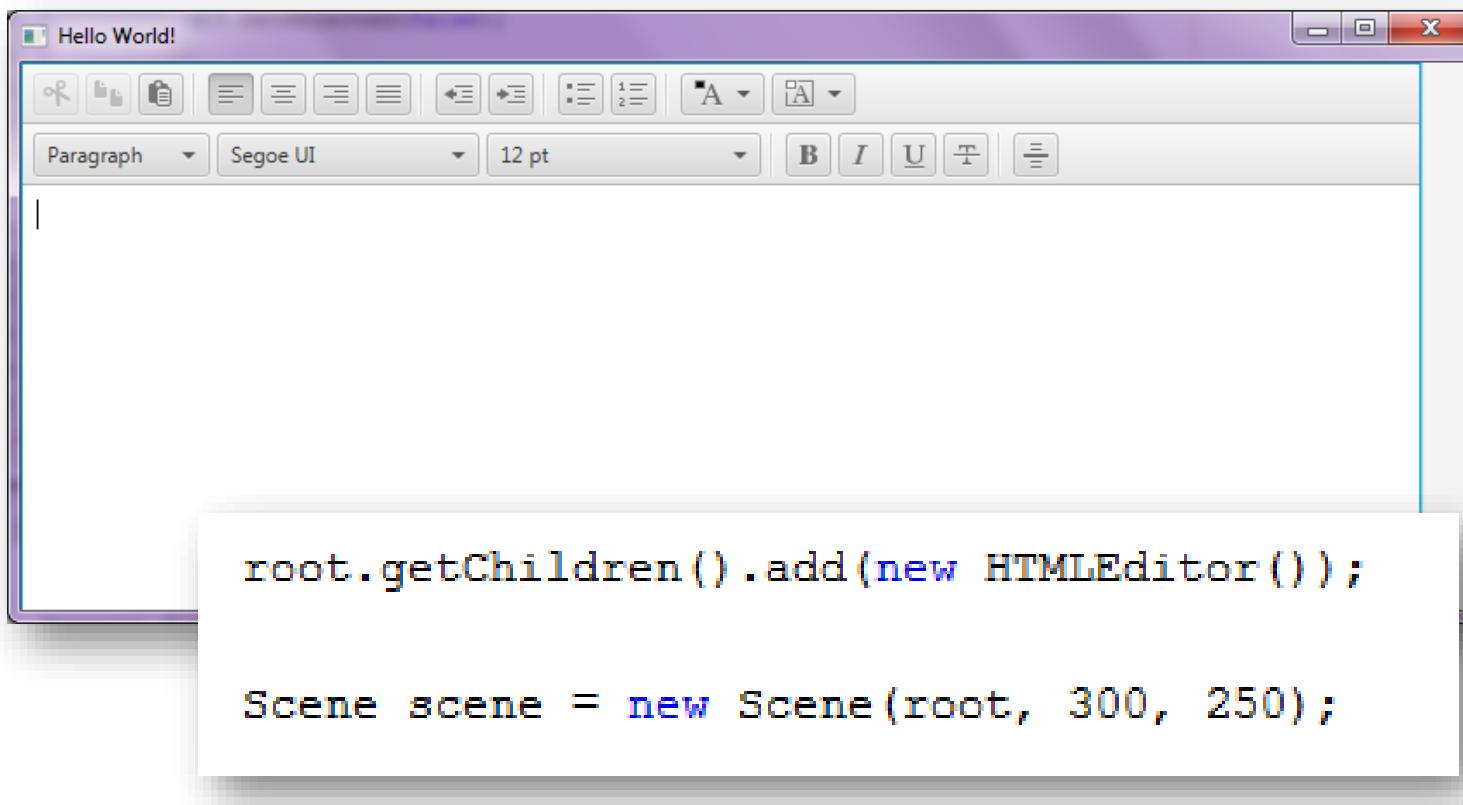
- Text input component that allows a user to enter a single line of unformatted text.
- As it is one single line **setPrefColCount()** to control the number of columns.
- TextField fires ActionEvent upon typing the Enter key.



Enter your last name.

```
TextField lastName = new TextField();
lastName.setPromptText("Enter your last name.");
```

- how to edit text in your JavaFX applications by using the embedded HTML editor.



# Creating Menus

- Constructing a Menu in JavaFX is not different than Swing, you create **MenuBar**, **Menu**, and **MenuItem**s, then we add them to each other.
- The difference between JavaFX and Swing that JavaFX does not have a pre-made Anchor for the menubar, so there is no **setMenuBar()** method like Swing.
- **MenuBar** it self is considered a node that can be added to any part of the located Pane.

- A **MenuBar** control traditionally is placed at the very top of the user interface, and embedded within it are **Menus**.
- To add a **Menu** to a **MenuBar** , you add it to the **menus Observable List**.

## Constructors

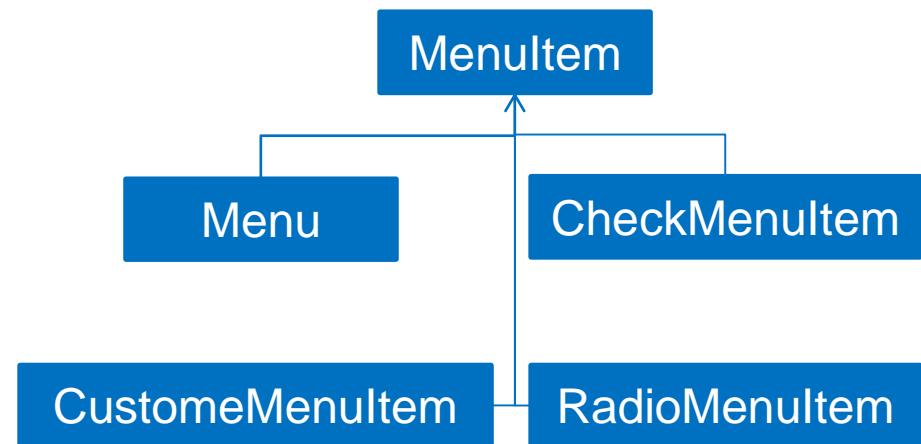
`MenuBar()`

`MenuBar(Menu...)`

## Methods

`ObservableList<Menu> getMenus()`

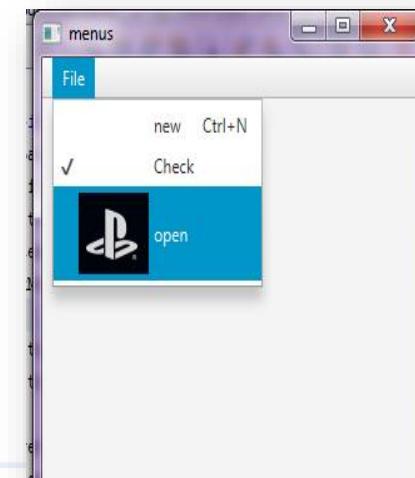
# Menus and MenuItem



- **MenuItem :**
  - to create one actionable option
  - The accelerator property enables accessing the associated action in one keystroke.
- **Menu :** to create a Menu / submenu
- **RadioButtonItem :** to create a mutually exclusive selection
- **CheckMenuItem :** to create an option that can be toggled between selected and unselected states

# Menus and MenuItem

```
public void start(Stage primaryStage) throws Exception {  
    MenuBar bar = new MenuBar();  
    Menu file = new Menu("File");  
  
    MenuItem newItem1 = new MenuItem("new");  
    newItem1.setAccelerator(KeyCombination.keyCombination("Ctrl+n"));  
  
    CheckMenuItem newItem2 = new CheckMenuItem("Check");  
  
    MenuItem openItem = new MenuItem("open");  
    openItem.setGraphic(new ImageView(new Image(getClass()  
        .getResourceAsStream("../img/icon.png"))));  
  
    file.getItems().addAll(newItem1, newItem2, openItem);  
    bar.getMenus().addAll(file);  
    BorderPane pane = new BorderPane();  
    pane.setTop(bar);  
    Scene scene = new Scene(pane, 300, 400);
```



# Building UI Using JavaFX

## Event Handling

- ❑ Event model in JavaFX is not different than Swing, there is an event source and a listener to the event.
- ❑ Unlike swing, JavaFX considers all event triggers as a reference property inside the **Node** class. We only need to link the correct event listener to the property we want to respond to.
- ❑ JavaFX uses only one generic interface to respond to all events **EventHandler<T extends Event>**.
- ❑ The only method in this interface is **handle(T Event)**.

- To handle the event using the event property reference.

```
Button b = new Button("click me !");
b.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.print("you clicked me...");
    }
});
```

- Using the **addEventHandler()** method.

```
Button b = new Button("click me !");
b.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent event) {
        //Event Handling code Here
    }
});
```

# Building UI Using JavaFX

## Layouts

- ❑ A JavaFX application can manually lay out the UI by setting the position and size properties for each UI element.
- ❑ JavaFX containers (**Panes**) are set of classes used to manage UI components positioning and size over the scene graph.
- ❑ layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes and the pane.
- ❑ All **panes** are **sub-class** of **Node** and they can be added to each other to form more complex layout.

# BorderPane

- ❑ BorderPane lays out children in top, left, right, bottom, and center positions.
- ❑ Only one node can be hosted at each position.
- ❑ The top and bottom children will be resized to their preferred heights and extend the width of the border pane.
- ❑ The left and right children will be resized to their preferred widths and extend the length between the top and bottom nodes.
- ❑ And the center node will be resized to fill the available space in the middle.
- ❑ BorderPane is commonly used as the root of a Scene.



# BorderPane

- ❑ listed below are the commonly used constructors and methods of this pane:

## Constructors

`BorderPane()`

`BorderPane(Node center)`

`BorderPane(Node center, Node top, Node right, Node bottom, Node left)`

`void setXXX(Node node)`

`Node getXXX()`

- ❑ **Note:** XXX will be replaced with one of the pane positions (Center, Left, Right, Top, Bottom).

- **FlowPane** lays out its children in a flow that wraps at the flowpane's boundary

```
FlowPane pane = new FlowPane();  
  
for (int i = 0; i < 5; i++) {  
    pane.getChildren().add(new Rectangle(100, 100,  
        new Color(new Random().nextDouble(),  
        new Random().nextDouble(),  
        new Random().nextDouble(), 1.0)));  
}  
}
```



- Nodes within the FlowPane can be oriented Horizontally or Vertically depending on the orientation property value.
  
- Spacing between nodes can be managed using the vgap, and hgap properties.
  
- To add nodes to a FlowPane we use the `getChildren()` method to get the node list of this container, then we use `add()`, or `addAll()` method to add nodes.

- ❑ listed below are the commonly used constructors and methods of this pane:

## Constructors

`FlowPane()`

`FlowPane(Node... children)`

`FlowPane(double hgap, double vgap, Node... children)`

`FlowPane(Orientation orientation, double hgap, double vgap)`

## Methods

`void setAlignment(Pos value)`

`void setHgap(double value)`

`void setVgap(double value)`

`ObservableList<Node> getChildren() ----> inherited from class Pane`

`void setOrientation(Orientation value)`

# GridPane

- GridPane lays out its children within a flexible grid of rows and columns.
- A child may be placed anywhere within the grid and may span multiple rows/columns.
- A child's placement within the grid is defined by it's layout constraints:

Constraint	Type	Description
columnIndex	integer	column where child's layout area starts.
rowIndex	integer	row where child's layout area starts.
columnSpan	integer	the number of columns the child's layout area spans horizontally.
rowSpan	integer	the number of rows the child's layout area spans vertically.

- If the **row/column** indices are not explicitly set, then the child will be placed in the first row/column.
- To add nodes to the GridPane we use the [add\(node, colIndex, rowIndex\)](#) method.

## Constructor

**GridPane()**

## Methods

**void addColumn(int columnIndex, Node... children)**

**void addRow(int rowIndex, Node... children)**

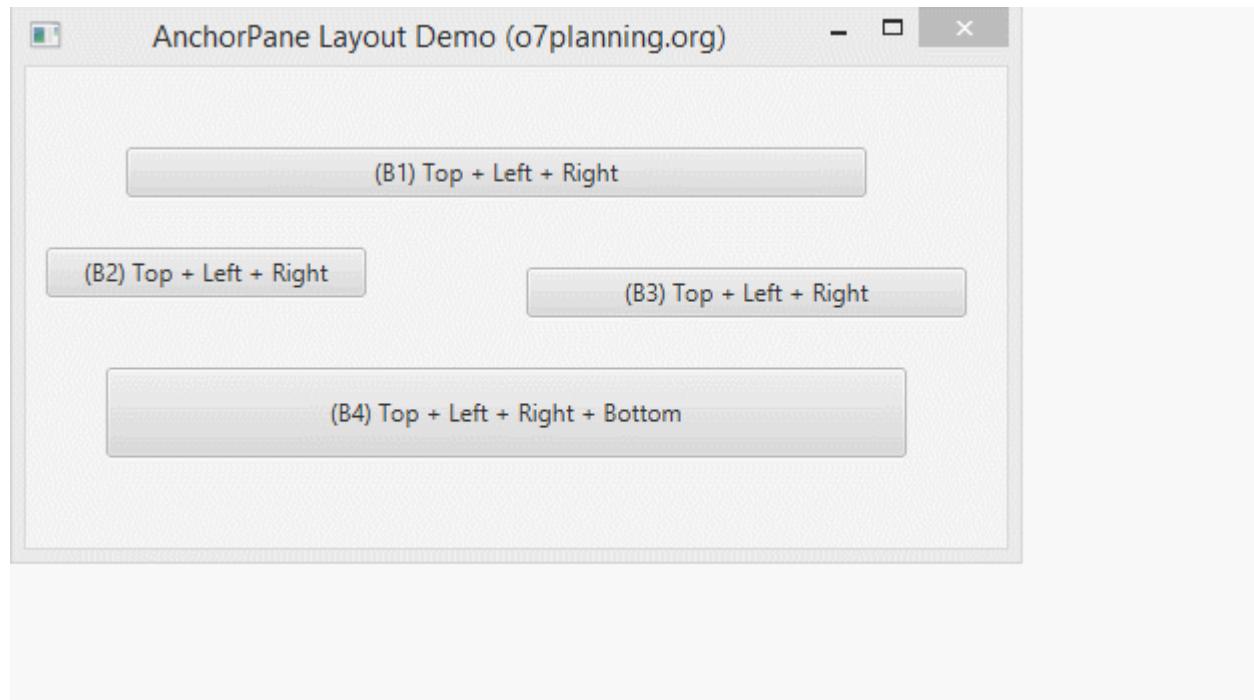
**void setHgap(double value)**

**void setVgap(double value)**

- AnchorPane allows the edges of child nodes to be anchored to an offset from the anchorpane's edges.
- If the anchorpane has a border and/or padding set, the offsets will be measured from the inside edge of those insets.
- AnchorPane has four constraints can be set for each child.

Constraint	type	Description
topAnchor	double	distance from the anchorpane's top insets to the child's top edge.
leftAnchor	double	distance from the anchorpane's left insets to the child's left edge.
bottomAnchor	double	distance from the anchorpane's bottom insets to the child's bottom edge.
rightAnchor	double	distance from the anchorpane's right insets to the child's right edge.

# AnchorPane



- The following are the commonly used methods of the AnchorPane:

## Constructors

AnchorPane()

AnchorPane(Node... children)

## Methods

**static void setBottomAnchor(Node child,Double value)**

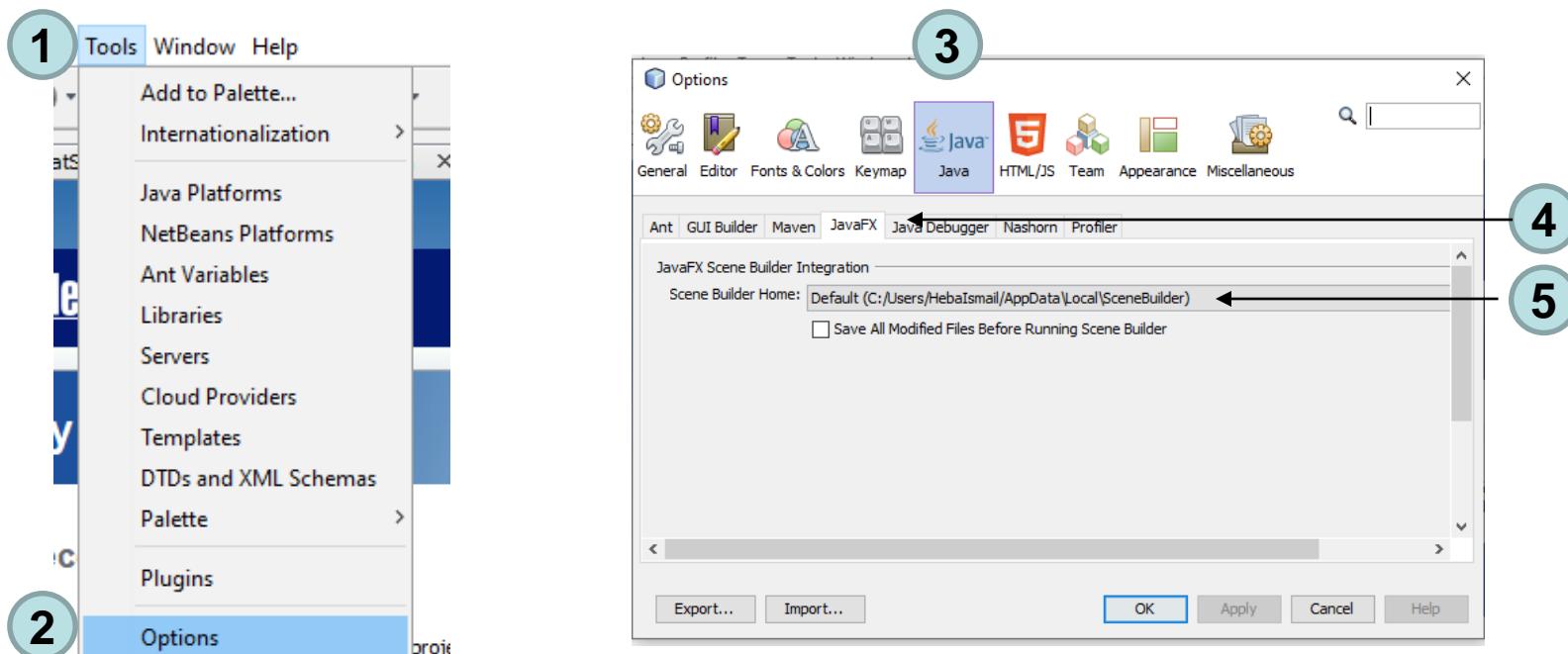
**static void setRightAnchor(Node child,Double value)**

**static void setLeftAnchor(Node child,Double value)**

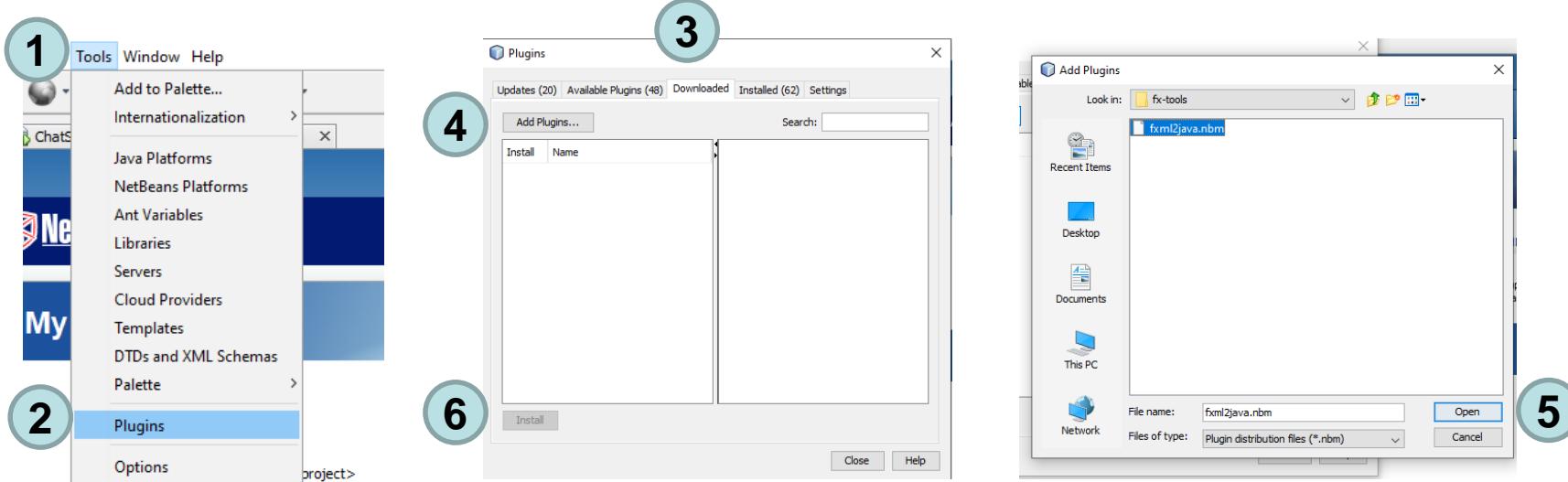
**static void setTopAnchor(Node child,Double value)**

# How to Setup Scene Builder

- Download & Setup Scene Builder ver 8.5.0.exe
- Make sure it's seen to your IDE by opening **tools menu** -> **Java** -> **JavaFX** -> check the path there that it contains the scene builder's path



- In NetBeans IDE open tools menu -> plugins -> downloaded tab -> add plugin -> open the directory where your .nbm file is located -> click open -> Install
- Your IDE will restart.



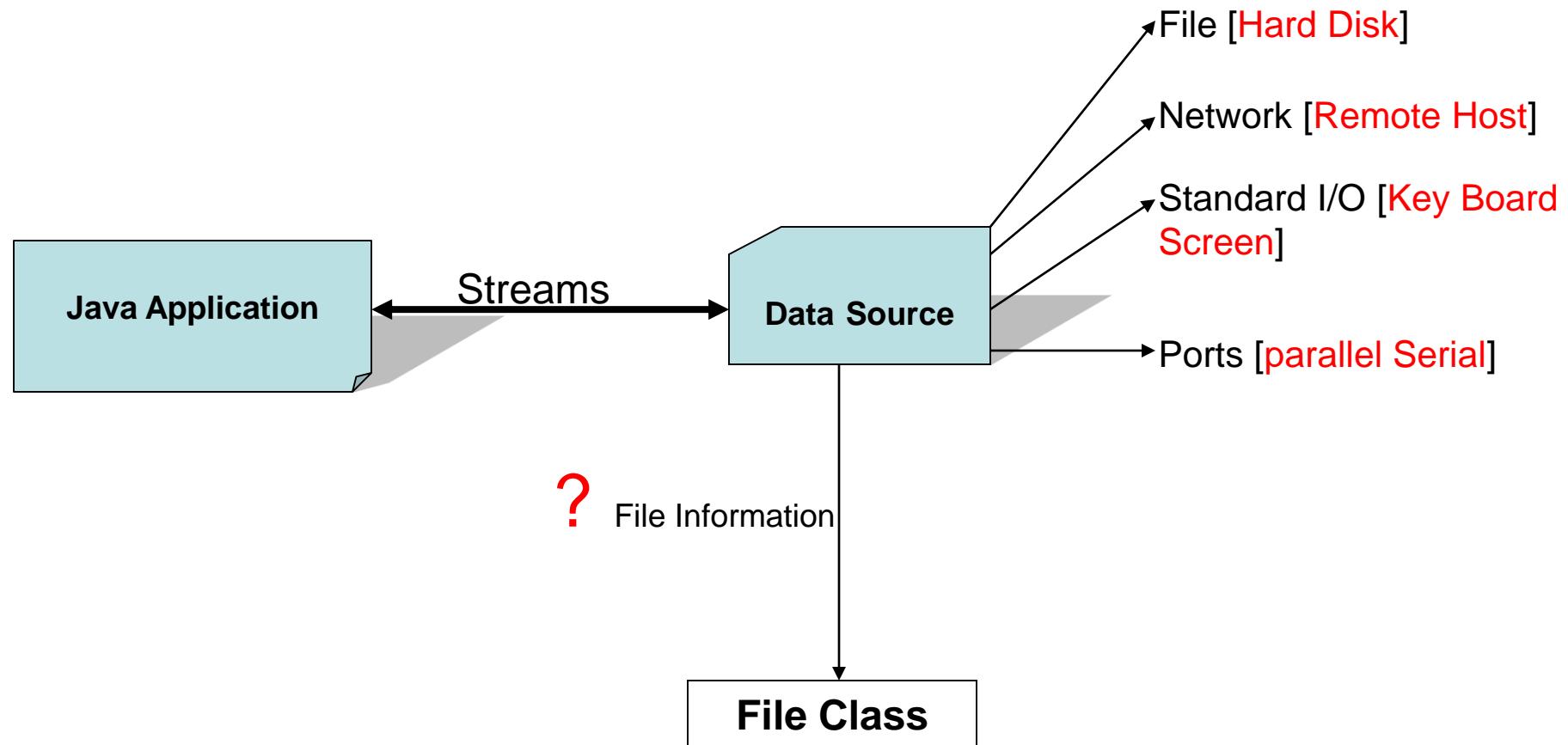
# 1. Text Editor Exercise

- Create a Frame-Based Application that carries out the basic functionalities of a simple text editor.
- The application should feature a text area and a menu bar with the following menus and menu items (with shortcuts):
  - **File:** New (ctrl+N), Open (ctrl+O), Save (ctrl+S), Exit (ctrl+E).
  - **Edit:** Cut, Copy, Paste, Delete, Select All.
  - **Help:** About.
  - **Note: File menu should contain: Save and Open using High and Low Level Streams**
- The About Dialog should display short information about the application and its version on multiple lines. It should also have an “Ok” button that disposes the dialog.

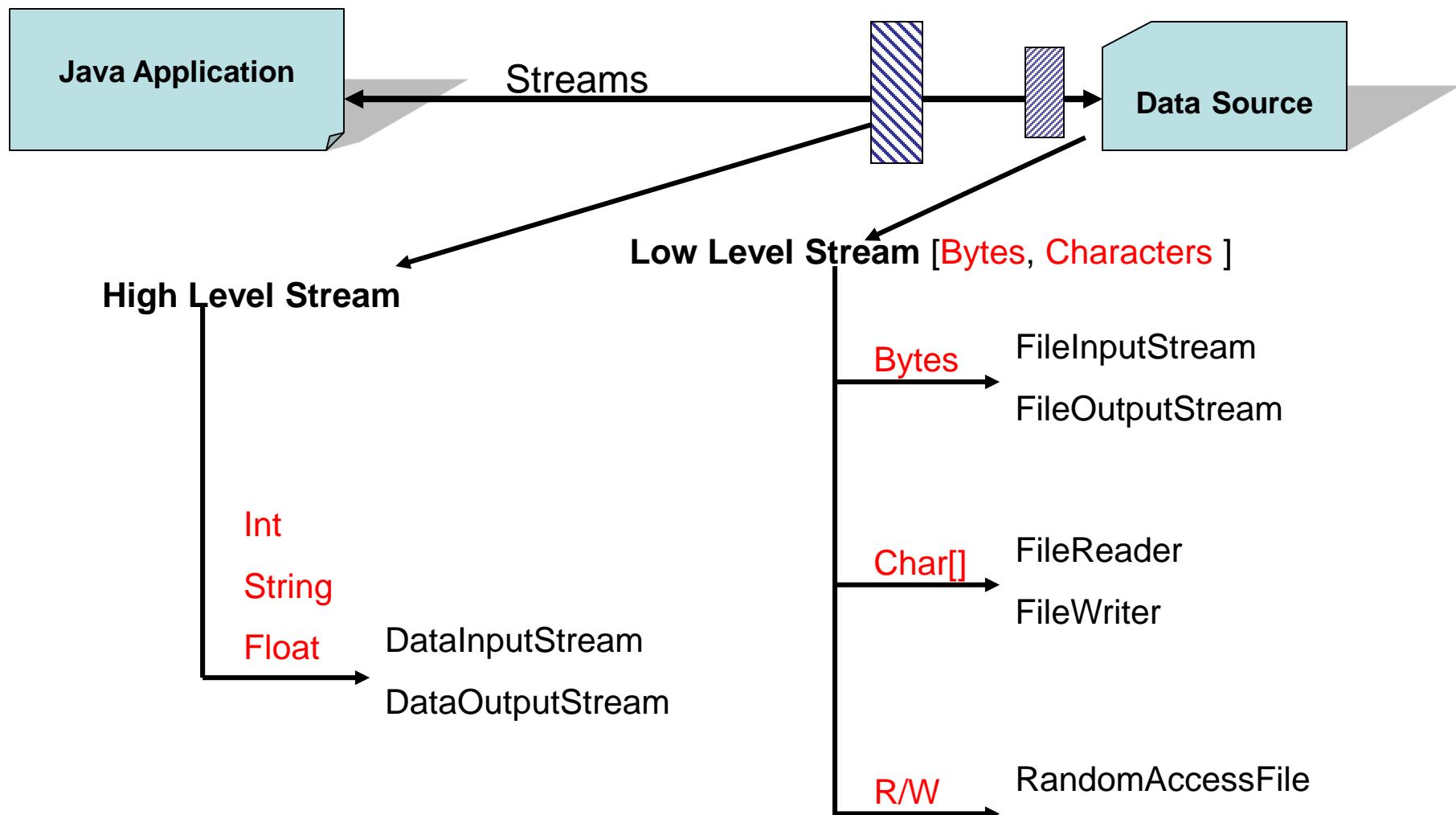
# Lesson 14

## Input and Output Streams

# Streams



# Streams



- A stream is a flow of data between a Data Source and a Data Sink (Destination).
- Streams are used for data input and output.
- An Input Stream is a stream that reads input into the application. Reading is **a blocking operation** (i.e. it blocks its thread).
- An Output Stream is a stream that carries out data from the application.
- Streams can be classified into two categories: Low Level Streams and High Level Streams.

- A Low Level Stream is a stream that is attached directly to the source/destination.
- It can only deal with raw data in the form of bytes or characters.

- A High Level Stream is a stream that is attached to a lower level stream (i.e. layered over it).
- It can deal with higher data types such as int, float, String, or even whole objects.
- A High Level Stream saves some conversion effort for the programmer. (e.g. Reading complete Strings instead of reading character by character then transferring them into a String).

- Commonly Used Constructor(s):
  - `File(String path)`
  - `File(String parent, String child)`
  - `File(File parent, String child)`
- Commonly Used Method(s):
  - `boolean exists()`
  - `boolean isFile()`
  - `boolean isDirectory()`
  - `String getName()`
  - `String getParent()`
  - `String getAbsolutePath()`

- Commonly Used Method(s):

- `String[] list()`
- `boolean canRead()`
- `boolean canWrite()`
- `boolean delete()`
- `long length()`
- `boolean createNewFile()`
- `boolean mkdir()`

- Commonly Used Constructor(s):
  - `FileInputStream(String file)`
  - `FileInputStream(File file)`
- Commonly Used Method(s):
  - `int read()`
  - `int read(byte[] b)`
  - `int read(byte[] b, int offset, int length)`
  - `int available()`
  - `long skip(long)`
  - `void close()`
- The `offset` parameter determines the start index in the destination array `b`.
- The `length` parameter specifies the maximum number of bytes to read.

- Commonly Used Constructor(s):
  - `FileOutputStream(String file)`
  - `FileOutputStream(File file)`
- Commonly Used Method(s):
  - `void write(int b)`
  - `void write(byte[] b)`
  - `void write(byte[] b, int offset, int length)`
  - `void close()`
  - `void flush()`
- The `offset` parameter determines the start index at the source array `b`.
- The `length` parameter specifies the number of bytes to write.

- Commonly Used Constructor(s):
  - `FileReader(String file)`
  - `FileReader(File file)`
- Commonly Used Method(s):
  - `int read()`
  - `int read(char[] c)`
  - `int read(char[] c, int offset, int length)`

- Commonly Used Constructor(s):
  - `FileWriter(String file)`
  - `FileWriter(File file)`
- Commonly Used Method(s):
  - `void write(int c)`
  - `void write(char[] c)`
  - `void write(char[] c, int offset, int length)`
  - `void write(String str)`
  - `void write(String str, int offset, int length)`

- Commonly Used Constructor(s):
  - `RandomAccessFile(String file, String mode)`
  - `RandomAccessFile(File file, String mode)`
- The `mode` parameter can assume the values “r” or “rw”.
- Commonly Used Method(s):
  - `long getFilePointer()`
  - `void seek(long position)`
  - `long length()`
  - `int read()`
  - `int read(byte[] b)`
  - `int read(byte[], int offset, int length)`
  - `void write(int b)`
  - `void write(byte[] b)`
  - `void write(byte[] b, int offset, int length)`

- Commonly Used Constructor(s):
  - `DataInputStream(InputStream in)`
- Commonly Used Method(s):
  - `int readInt()`
  - `long readLong()`
  - `float readFloat()`
  - `double readDouble()`
  - `String readUTF()`

- Commonly Used Constructor(s):
  - `DataOutputStream(OutputStream out)`
- Commonly Used Method(s):
  - `void writeInt(int i)`
  - `void writeLong(long l)`
  - `void writeFloat(float f)`
  - `void writeDouble(double d)`
  - `void writeUTF(String str)`

- The following code sample is for allowing the user to choose a text file to open in a text area:

```
class MyOpenListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        FileChooser fc = new FileChooser();
        File file = fc.showOpenDialog();
        if(file != null)
        {
            FileInputStream fis = new FileInputStream(file);
            int size = fis.available();
            byte[] b = new byte[size];
            fis.read(b);
            ta.setText(new String(b));
            fis.close();
        }
    }
}
```

- The following code sample is for allowing the user to choose a text file to save the text he entered in a text area:

```
class MySaveListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        FileChooser fc = new FileChooser();
        File file = fc.showSaveDialog();
        if(file != null)
        {
            FileOutputStream fos = new FileOutputStream(file);
            byte[] b = ta.getText().getBytes();
            fos.write(b);
            fos.close();
        }
    }
}
```

# Lab Exercise

# 1. Text Editor Exercise

- Create a Frame-Based Application that carries out the basic functionalities of a simple text editor.
- The application should feature a text area and a menu bar with the following menus and menu items (with shortcuts):
  - **File:** New (ctrl+N), Open (ctrl+O), Save (ctrl+S), Exit (ctrl+E).
  - **Edit:** Cut, Copy, Paste, Delete, Select All.
  - **Help:** About.
  - **Note:** File menu should contain: Save and Open using High and Low Level Streams
- The About Dialog should display short information about the application and its version on multiple lines. It should also have an “Ok” button that disposes the dialog.

# Java Collections

# 3 overloaded uses of the word "collection"

- **collection** (lowercase c), which represents any of the data structures in which objects are stored and iterated over.
- **Collection** (capital C), which is actually the `java.util.Collection` interface from which `Set`, `List`, and `Queue` extend. (That's right, extend, not implement. There are no direct implementations of `Collection`.)

- **Collections** (capital C and ends with s) is the `java.util.Collections` class that holds a pile of static utility methods for use with collections.

- The **Java collections** framework is a set of **generic types** that you use to create **collection classes** that support various ways for you to store and manage objects of any kind in memory.
- **A collection class** is simply a class that organizes a set of objects of a given type in a particular way, such as in a linked list or a pushdown stack.
- The majority of types that make up the collections framework are defined in the `java.util` package.

- There are three main types of collections that organize objects in different ways, called **sets**, **sequences**, and **maps**.
- In Java, collections store references only—the objects themselves are external to the collection.

# Flavors of Collections

## Sequences

### Sets

- Unique things (classes that implement Set).

### Lists

- Lists of things (classes that implement List).

### Queues

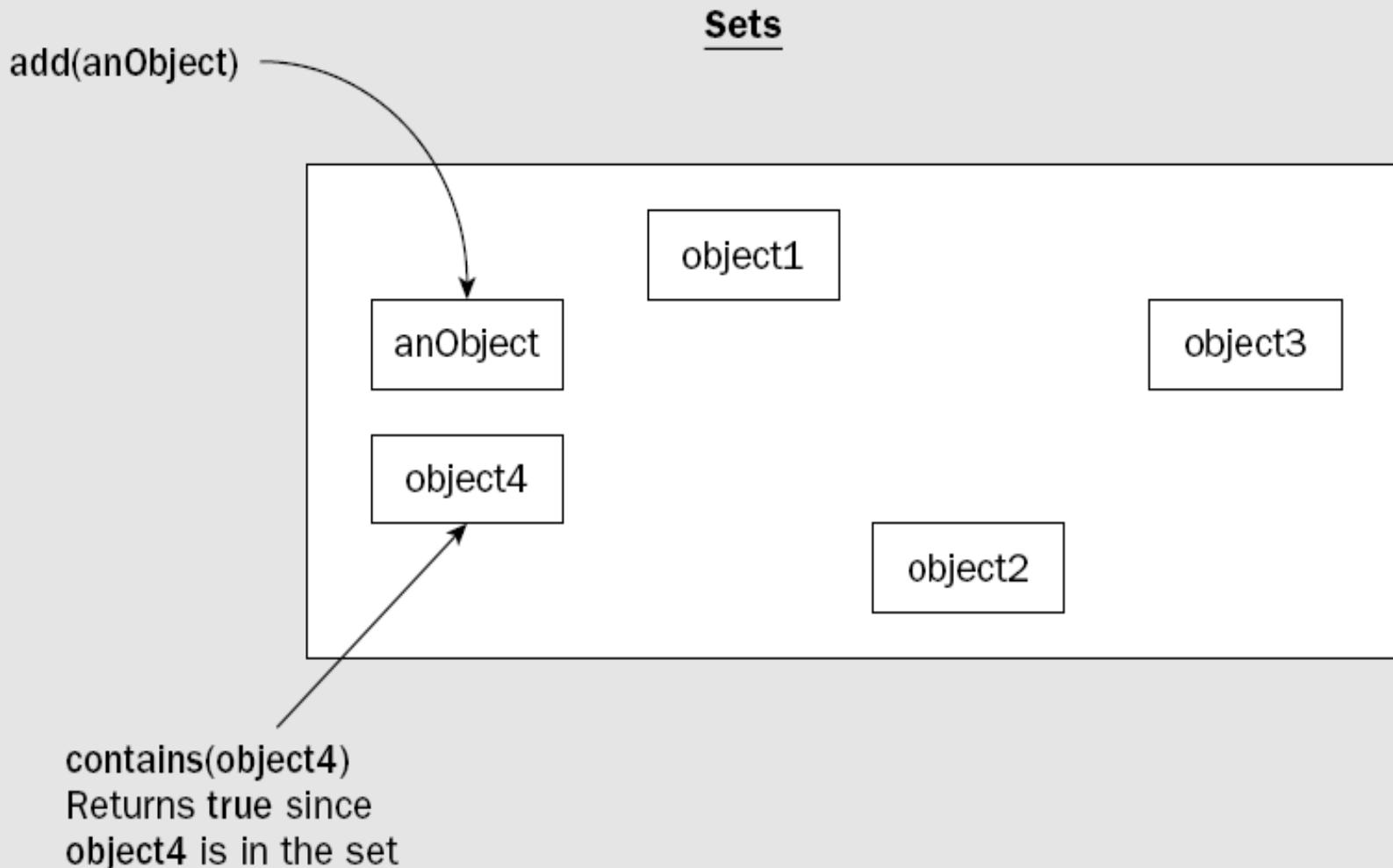
- Things arranged by the order in which they are to be processed.

### Maps

- Things with a unique ID (classes that implement Map).

- A set is probably the simplest kind of collection you can have.
- A Set cares about uniqueness, it doesn't allow duplicates.
- Here, the objects are not ordered in any particular way at all, and objects are simply added to the set without any control over where they go.
- The Following Figure illustrates the idea of a set.

# Collection Framework – Sets cont'd



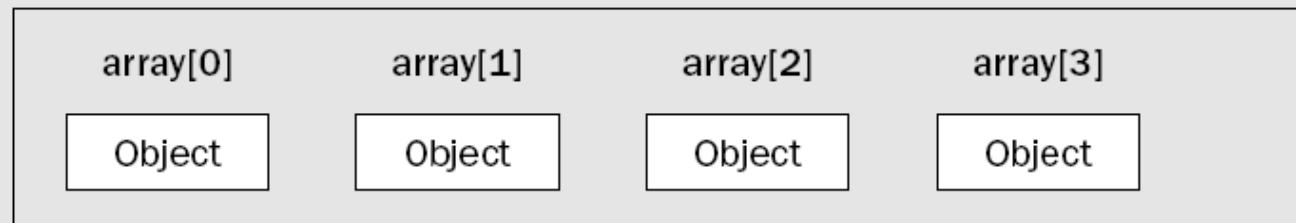
- You can add objects to a set and iterate over all the objects in a set.
- You can also check whether a given object is a member of the set or not.
- For this reason you cannot have duplicate objects in a set—each object in the set must be unique.

	HashSet	LinkedHashSet	TreeSet
Sorted	<ul style="list-style-type: none"><li>No</li></ul>	<ul style="list-style-type: none"><li>No</li></ul>	<ul style="list-style-type: none"><li>Yes</li></ul>
Ordered	<ul style="list-style-type: none"><li>No</li></ul>	<ul style="list-style-type: none"><li>Elements are in the order in which they were inserted.</li></ul>	<ul style="list-style-type: none"><li>Elements will be in ascending order, according to natural order.</li></ul>
Advantages	<ul style="list-style-type: none"><li>Better performance with efficient hashCode()</li></ul>	<ul style="list-style-type: none"><li>Maintains a doubly-linked List across all elements.</li></ul>	<ul style="list-style-type: none"><li>You can use a constructor takes Comparator</li></ul>

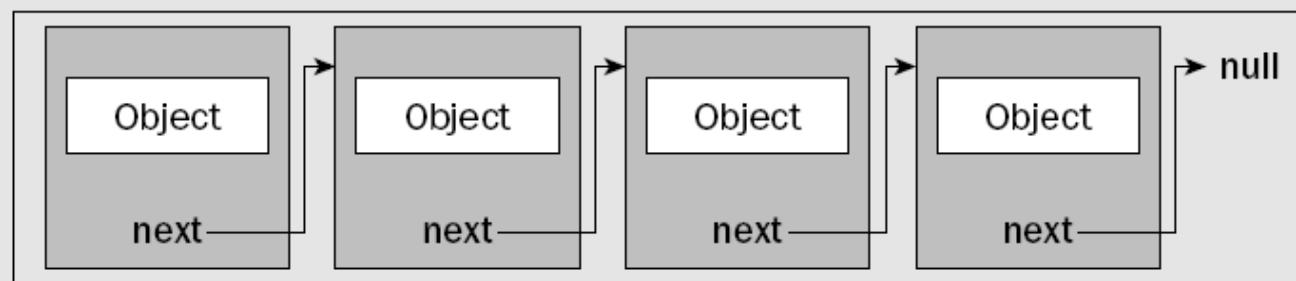
- In HashSet and LinkedHashSet, you must override hashCode() or , the default Object hashCode( ) method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.

- A primary characteristic of a sequence is that the objects are stored in a linear fashion, not necessarily in any particular order, but organized in an arbitrary fixed sequence with a beginning and an end.
- The Following Figure illustrates the organization of objects in the various types of sequence collections that you have available.

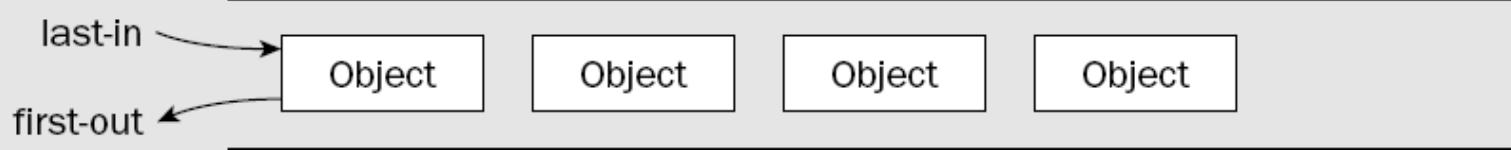
Array or Vector



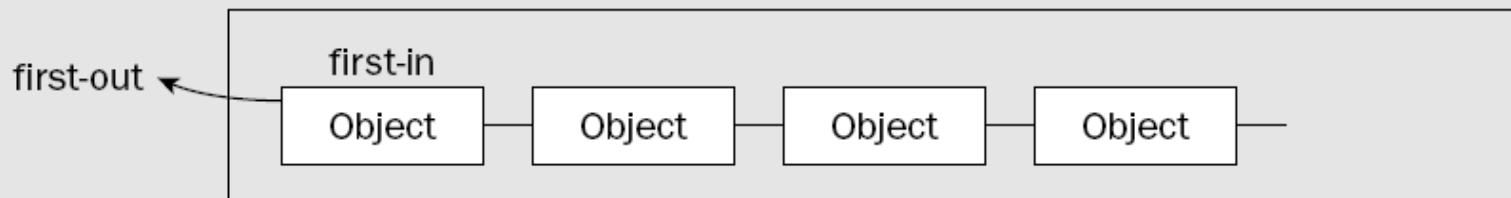
Linked List



Stack



Queue



- Because a sequence is linear, you will be able to add a new object only at the beginning or at the end, or insert a new object following a given object position in the sequence.
- In the Java collections framework, types that define sequences are subdivided into two subgroups, **lists** and **queues**.
- **Vectors, linked lists, and stacks** are all lists.

- A List cares about the index.
- The one thing that List has more than the non-list is a set of methods related to the index. Those key methods include things like:
  - `get(int index)`, `indexOf(Object o)`, `add(int index, Object obj)`, and so on.
- All three List implementations are ordered by index position.
- The position can be determined either by:
  - setting an object at a specific index or
  - by adding it without specifying position, in which case the object is added to the end.

# List Implementations

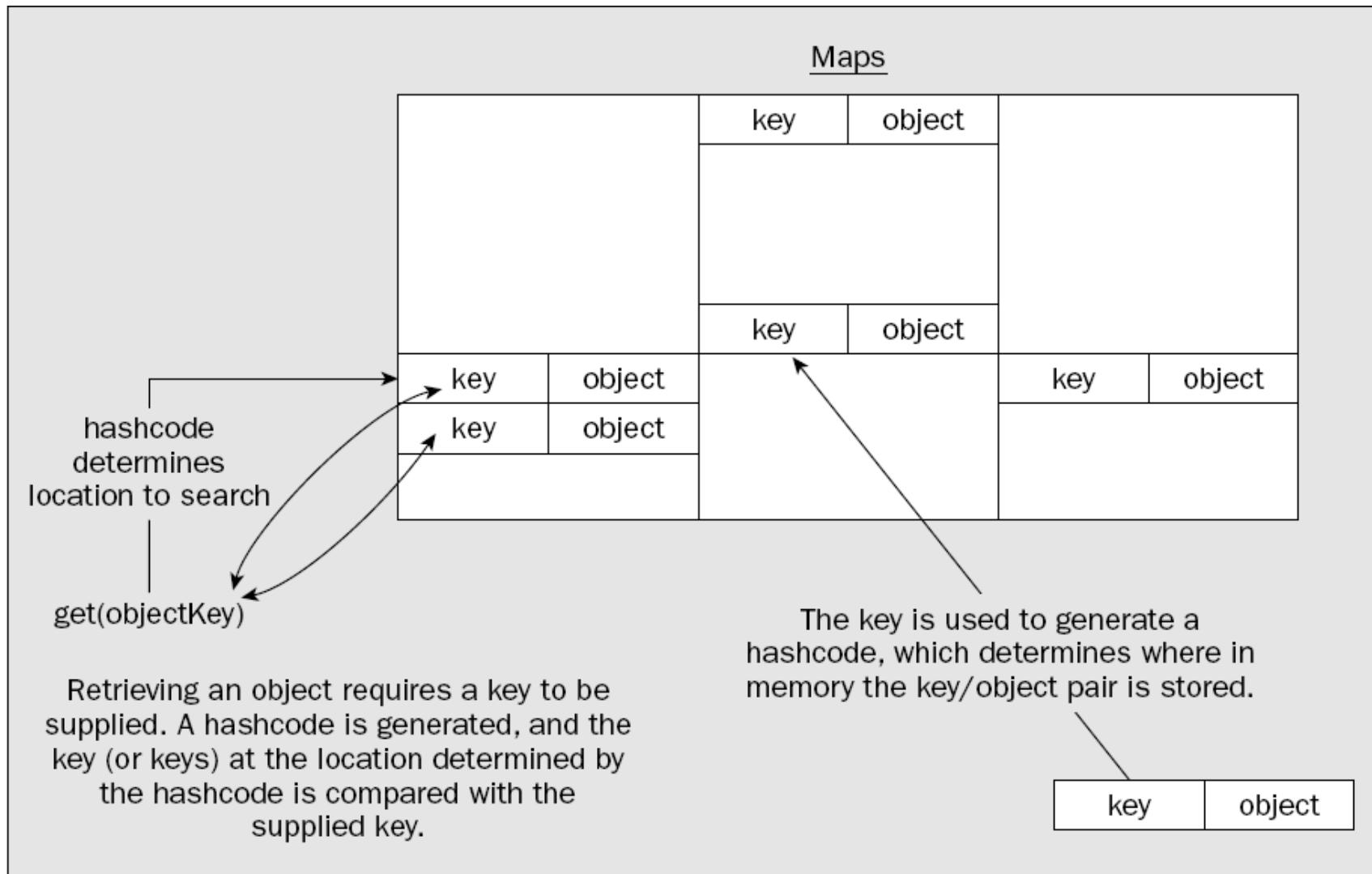
Vector	ArrayList	LinkedList
<ul style="list-style-type: none"><li>• Methods are synchronized for thread safety.</li></ul>	<ul style="list-style-type: none"><li>• Better iteration performance than LinkedList.</li><li>• Better than Vector if synchronization is not needed.</li></ul>	<ul style="list-style-type: none"><li>• Elements are doubly-linked to one another that makes it an easy choice for implementing a stack or queue.</li><li>• Good choice when you need fast insertion and deletion.</li></ul>

- **Queue** is designed to apply the FIFO.
- **PriorityQueue** is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue.
  - Elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first)
  - or according to a Comparator.
  - Elements' ordering represents their relative priority.

- A map is rather different from a set or a sequence collection because each entry involves a pair of objects.
- A map is also referred to sometimes as a dictionary because of the way it works.
- Each object that is stored in a map has an associated key object, and the object and its key are stored together as a pair.

- **The key** determines where the object is stored in the map, and when you want to retrieve an object, you must supply the appropriate key.
- The Following Figure shows how a map works.

# Collection Framework – Maps cont'd



# Maps Implementations

	HashMap	Hashtable
Sorted	<ul style="list-style-type: none"><li>• No</li></ul>	<ul style="list-style-type: none"><li>• No</li></ul>
Ordered	<ul style="list-style-type: none"><li>• No</li></ul>	<ul style="list-style-type: none"><li>• Elements are in the order in which they were inserted.</li></ul>
Advantages	<ul style="list-style-type: none"><li>• Better performance with efficient hashCode( )</li></ul>	<ul style="list-style-type: none"><li>• Methods are synchronized for thread safety.</li><li>• No Null Keys or Objects</li></ul>

# Maps Implementations cont'd

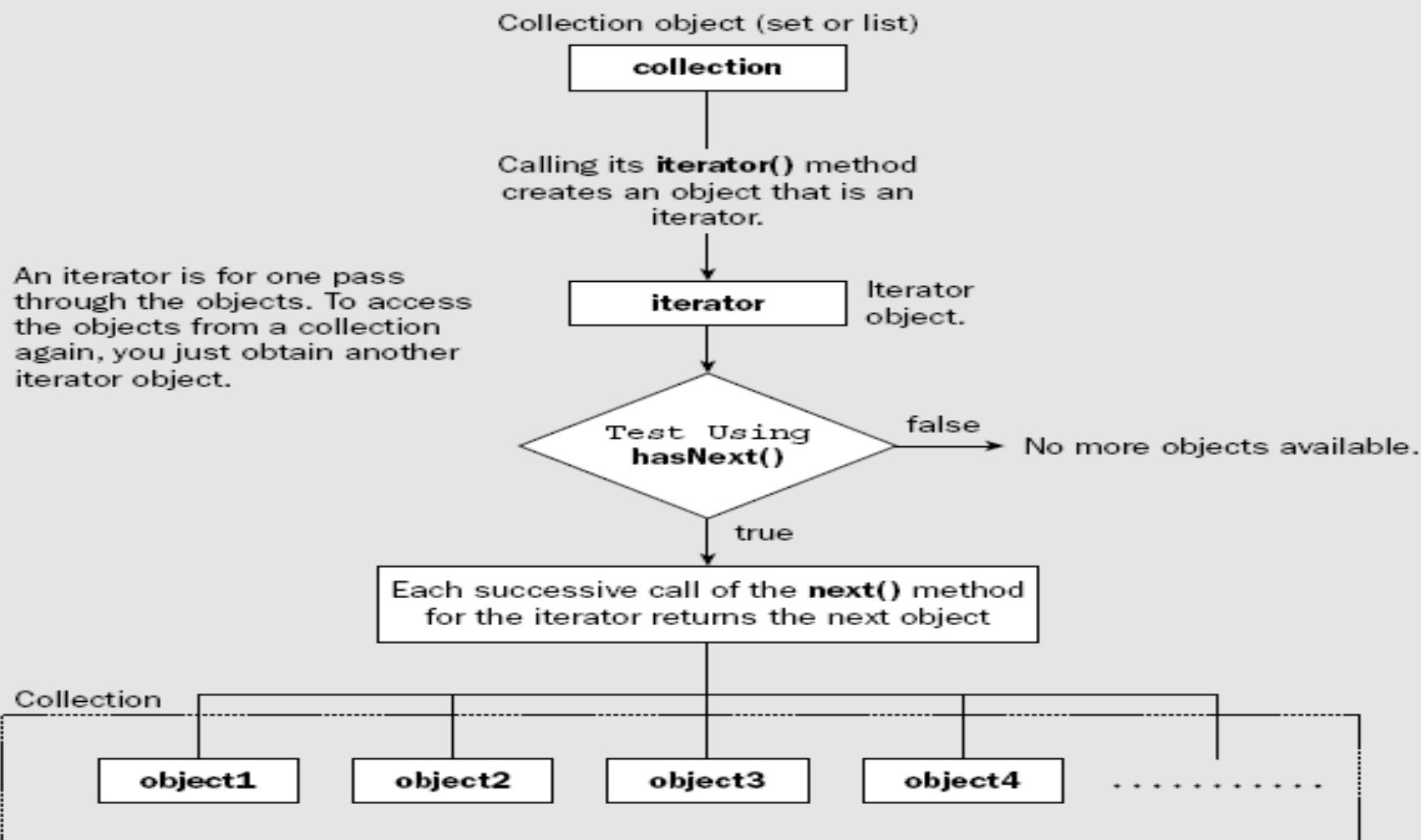
	LinkedHashMap	TreeMap
Sorted	<ul style="list-style-type: none"><li>No</li></ul>	<ul style="list-style-type: none"><li>sorted by the natural order or custom comparison of elements</li></ul>
Ordered	<ul style="list-style-type: none"><li>Elements are in the order in which they were inserted.</li></ul>	<ul style="list-style-type: none"><li>Yes.</li></ul>
Advantages	<ul style="list-style-type: none"><li>Faster iteration performance than HashMap</li></ul>	

- An Iterator is an object that you can use once to retrieve all the objects in a collection one by one.
- Using an **Iterator is a standard mechanism** for accessing each of the elements in a collection.
- Any collection object that represents a set or a sequence can create an object of type Iterator that behaves as an iterator.
- Types representing maps do not have methods for creating iterators.

- A map class provides methods to enable the keys or objects, or indeed the key/object pairs, to be viewed as a set, so you can then obtain an iterator to iterate over the objects in the set view of the map.
- An Iterator object encapsulates references to all the objects in the original collection in some sequence, and they can be accessed one by one using the Iterator interface methods

# Iterator cont'd

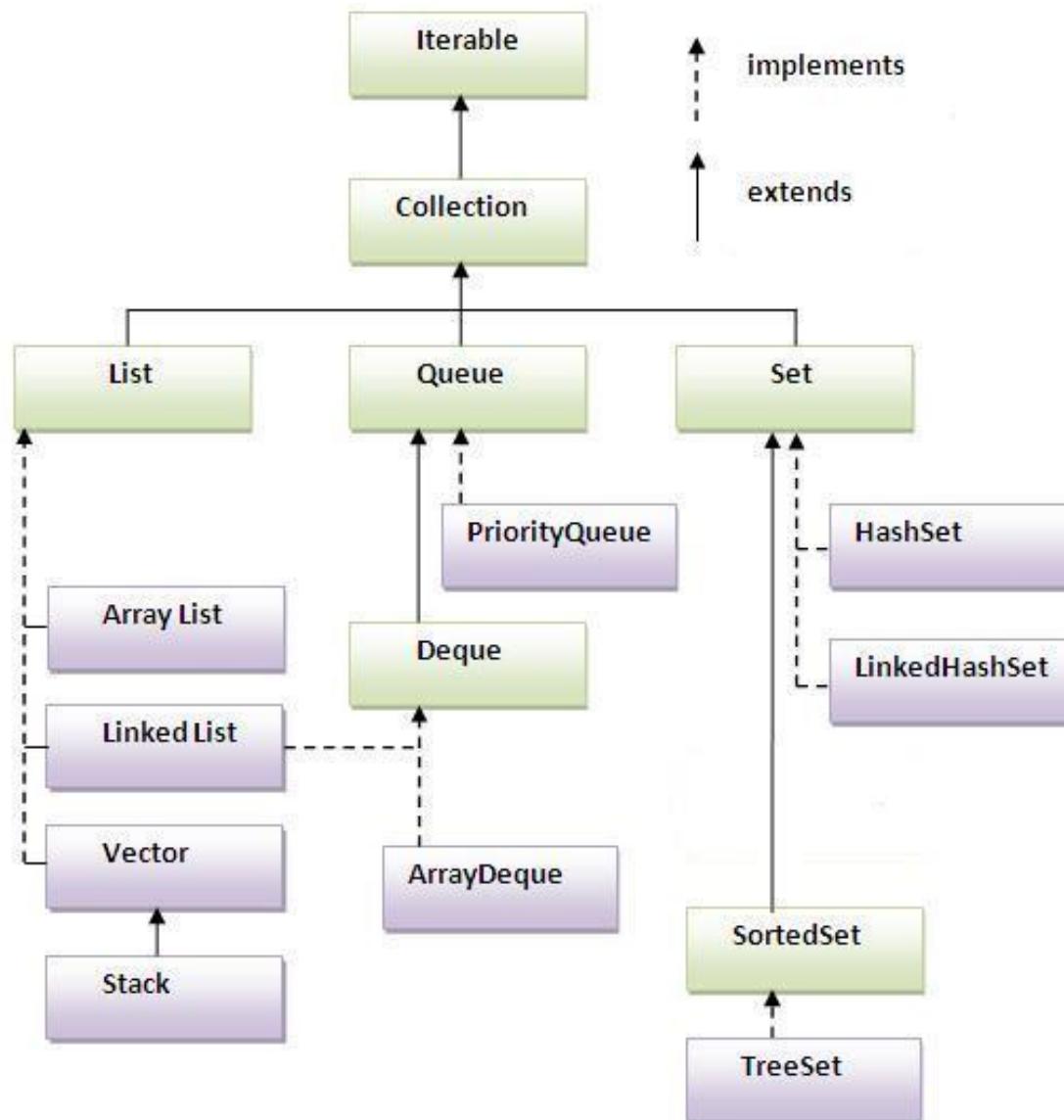
- The basic mechanism for using an iterator is illustrated in the figure below:



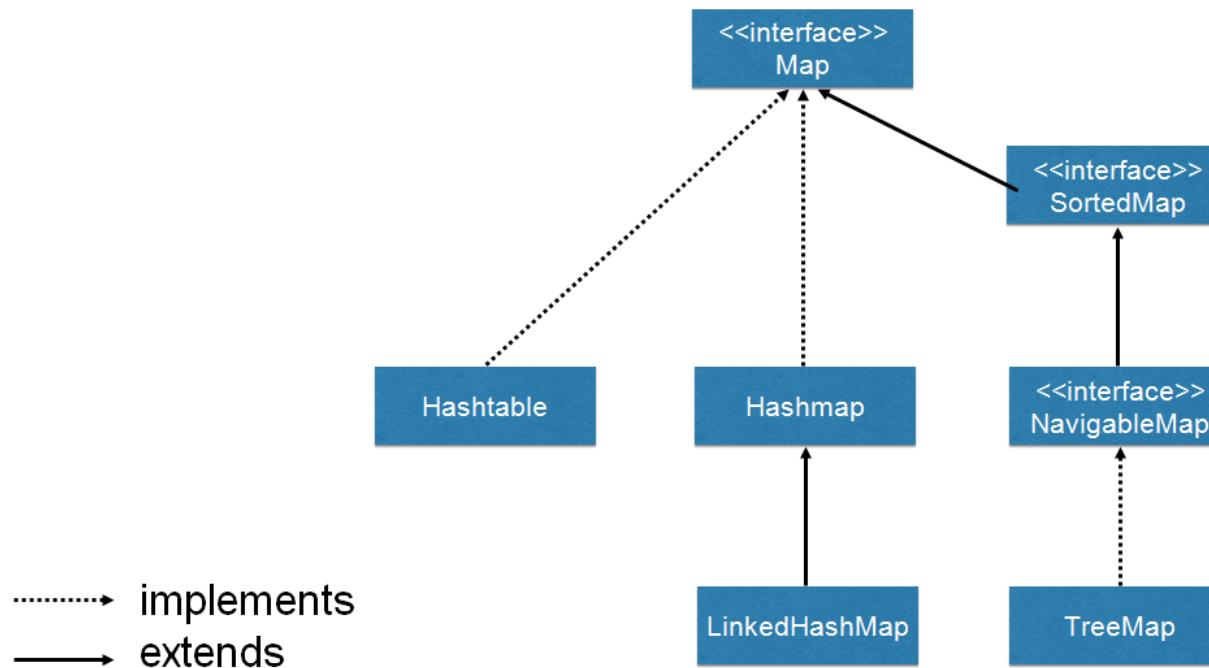
- Methods of Iterator interface :

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator.

# Collection Classes [java.util]



# Map Interface



- Generics allow you to abstract over types. The most common examples are container types, such as those in the Collections hierarchy.
- Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1  
myIntList.add(new Integer(0)); // 2  
Integer x = (Integer) myIntList.iterator().next(); // 3
```

- The Cast in line three is annoying, It also introduces the possibility of a run time error, since the programmer may be mistaken.

- What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); //1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); //3'
```

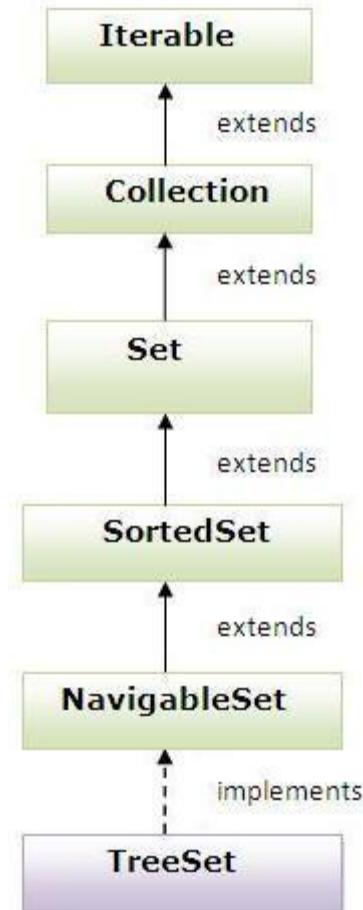
- Notice the type declaration for the variable myList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>.
- The compiler can now check the type correctness of the program at compile-time.
- In contrast, the cast tells us something the programmer thinks is true at a single point in the code.
- The net effect, especially in large programs, is improved readability and robustness

- In Java SE 7,
  - You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>).
  - This pair of angle brackets, <>, is informally called *the diamond*.
  - For example:

```
Box<Integer> integerBox = new Box<>();
```

- Java TreeSet class

- contains unique elements only
- maintains ascending order.

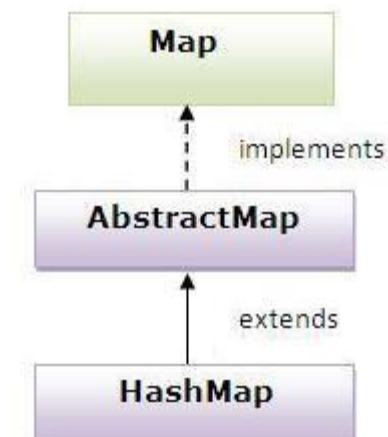


# Example

```
import java.util.*;  
  
class TestCollection11{  
  
    public static void main(String args[]){  
  
        TreeSet<String> al=new TreeSet<String>();  
        al.add("Ravi");  
        al.add("Vijay");  
        al.add("Ravi");  
        al.add("Ajay");  
  
        Iterator<String> itr=al.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next());  
        }  
    }  
}
```

Output:  
Ajay  
Ravi  
Vijay

- Java HashMap class
  - A HashMap contains values based on the key.
  - It implements the Map interface and extends AbstractMap class.
  - It may have one null key and multiple null values.
  - It contains only unique elements.
  - It maintains no order.



# Example

```
import java.util.*;  
class TestCollection13{  
    public static void main(String args[]){  
  
        HashMap<Integer,String> hm=new HashMap<Integer,String>();  
  
        hm.put(100,"Amit");  
        hm.put(101,"Vijay");  
        hm.put(102,"Rahul");  
  
        for(Map.Entry m:hm.entrySet()){  
            System.out.println(m.getKey()+" "+m.getValue());  
        }  
    }  
}
```

Output:  
102 Rahul  
100 Amit  
101 Vijay

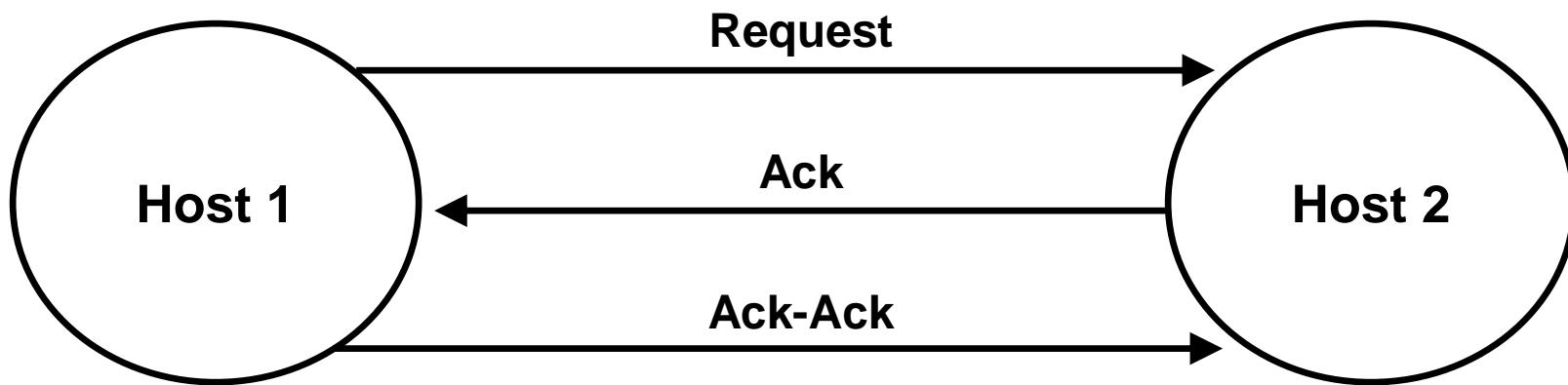
# Lesson 15

## Networking in Java

# Overview of TCP and UDP Protocols

TCP	UDP
Connection Oriented (Handshaking procedure)	Connection Less
Continuous Stream	Message Oriented
Reliable (Error Detection)	Unreliable

# 3-Way Handshaking Procedure



- In order to connect to a remote host, two pieces of information are essentially required:
  - IP Address (of remote machine)
  - Port Number (to identify the service at the remote machine)
- **Socket = Address + Port**
- Range of port numbers: 0 → 65,535
- From 0 → 1,024 are reserved for well known services, such as:
  - HTTP: 80
  - FTP: 21
  - Telnet: 23
  - SMTP: 25

Server	Client
<b>1. Create a server socket (bind the service to a certain port)</b>	
<b>2. Listen for connections</b>	<b>1. Create a socket (connect to the server)</b>
<b>3. Accept connection and transfer the client request to a virtual port.</b>	
<b>4. Obtain input and output streams</b>	<b>2. Obtain input and output streams</b>
<b>5. Send and receive data</b>	<b>3. Send and receive data.</b>
<b>6. Terminate connection (after communication has ended)</b>	<b>4. Terminate connection (after communication has ended)</b>

- Commonly Used Constructor(s):
  - `ServerSocket(int port)`
  - `ServerSocket(int port, int maxCon)`
- Commonly Used Method(s):
  - `Socket accept()`
  - `close()`

- Commonly Used Constructor(s):
  - `Socket(String address, int port)`
  - `Socket(InetAddress address, int port)`
- Commonly Used Method(s):
  - `InputStream getInputStream()`
  - `OutputStream getOutputStream()`

- InetAddress class has no public constructor.
- Commonly Used Method(s):
  - `static InetAddress getByName(String host)`
  - `static InetAddress[] getAllByName(String host)`
  - `static InetAddress getLocalHost()`
  - `String getHostName()`
  - `String getHostAddress()`
  - `Byte[] getAddress()`

# Simple Client/Server Console Example

## Server Application

- The following code sample is for creating a simple one-to-one client/server application, where each machine sends out a string and receives a string:

```
public class Server
{
    ServerSocket myServerSocket;
    Socket s;
    DataInputStream dis ;
    PrintStream ps;
    public static void main(String[] args)
    {
        new Server();
    }

    public Server()
    {
        try
        {
            myServerSocket = new ServerSocket(5005);
            s = myServerSocket.accept ();
            dis = new DataInputStream(s.getInputStream ());
            ps = new PrintStream(s.getOutputStream ());
        }
    }
}
```

# Simple Client/Server Console Example

## Server Application cont'd

```
        String msg = dis.readLine();
        System.out.println(msg);
        ps.println("Data Received");
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
    finally
    {
        try
        {
            ps.close();
            dis.close();
            s.close();
            myServerSocket.close();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

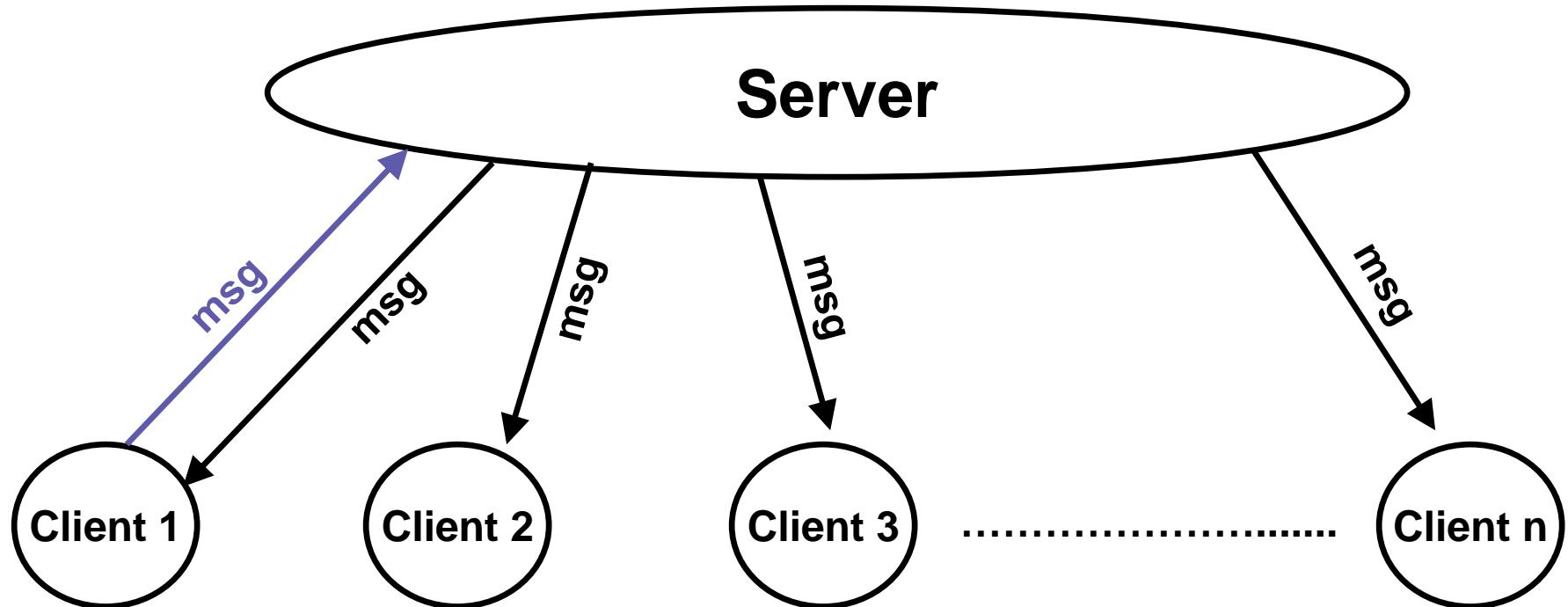
```
public class Client
{
    Socket mySocket;
    DataInputStream dis ;
    PrintStream ps;
    public static void main(String[] args)
    {
        new Client();
    }

    public Client()
    {
        try
        {
            mySocket = new Socket("127.0.0.1", 5005);
            dis = new DataInputStream(mySocket.getInputStream ());
            ps = new PrintStream(mySocket.getOutputStream ());
            ps.println("Test Test");
            String replyMsg = dis.readLine();
        }
    }
}
```

# Simple Client/Server Console Example

## Client Application cont'd

```
        System.out.println(replyMsg);
    }
catch(IOException ex)
{
    ex.printStackTrace();
}
finally
{
    try
    {
        ps.close();
        dis.close();
        mySocket.close();
    }
catch(Exception ex)
{
    ex.printStackTrace();
}
}
}
```



- The following code sample represents a server side application that holds a chat room that several clients can connect to and chat with each other. The server application is divided into two classes: ChatServer and ChatHandler:

```
public class ChatServer
{
    ServerSocket serverSocket;
    public ChatServer()
    {
        serverSocket = new ServerSocket(5005);
        while(true)
        {
            Socket s = serverSocket.accept();
            new ChatHandler(s);
        }
    }
    public static void main(String[] args)
    {
        new ChatServer();
    }
}
```

```
class ChatHandler extends Thread
{
    DataInputStream dis;
    PrintStream ps;
    static Vector<ChatHandler> clientsVector =
        new Vector<ChatHandler>();

    public ChatHandler(Socket cs)
    {
        dis = new DataInputStream(cs.getInputStream());
        ps = new PrintStream(cs.getOutputStream());
        ChatHandler.clientsVector.add(this);
        start();
    }
}
```

```
public void run()
{
    while(true)
    {
        String str = dis.readLine();
        sendMessageToAll(str);
    }
}

void sendMessageToAll(String msg)
{
    // for(ChatHandler ch : clientsVector)
    for(int i=0 ; i<clientsVector.size() ; i++)
    {
        clientsVector[i].ps.println(msg);
    }
}
```

- General Guidelines for building the client side GUI application:
  - Construct GUI and Frame Layout.
  - Create Socket and Streams
  - Register the Send Button Listener
  - Create and start the Reader Thread

# Lab Exercise

- Write the simple client/server application.
- Complete the GUI client side application of the Chat room Application.
- Remember to handle all thrown exceptions and to replace any deprecated methods with new ones.

# Create A GUI Desktop Application

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

