Java™ Education & Technology Services

Java Programming

Lesson 1

Introduction To Java

Brief History of Java

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

Brief History of Java

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

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

Java Principles

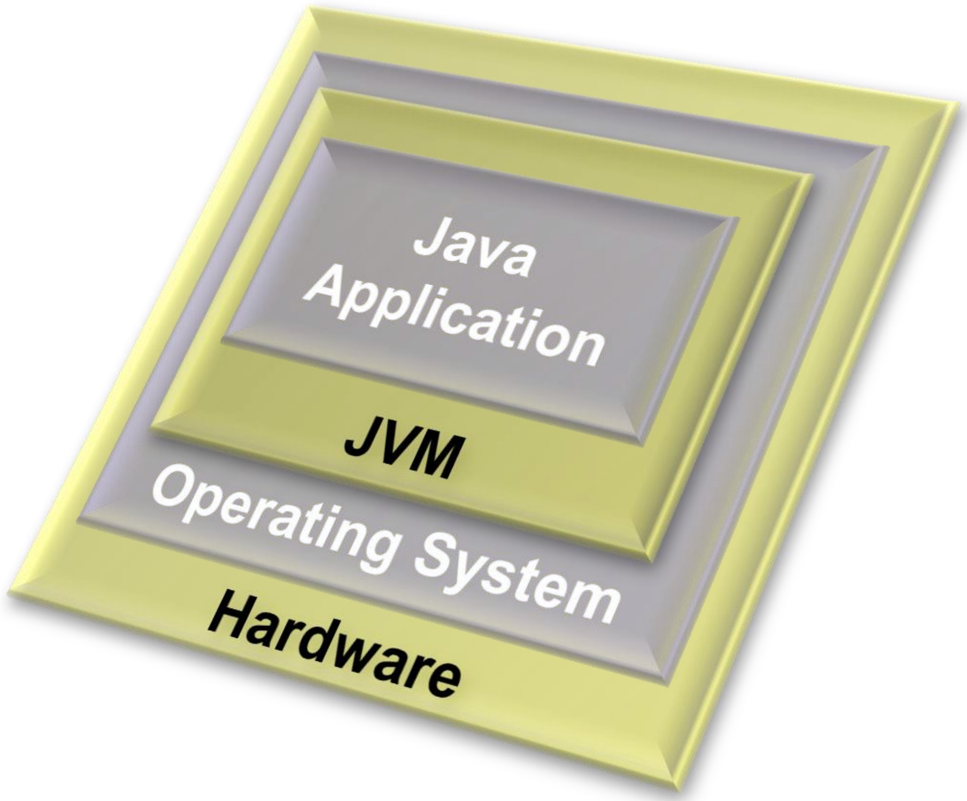
- Primary goals in the design of the Java programming language:
  - **Simple, object oriented, and easy to learn**.
  - **Robust and Secure**.
  - **Architecture neutral and portable**.
  - **Compiled and Interpreted.**
  - **Multithreaded**.
  - **Networked.**

Java Features

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

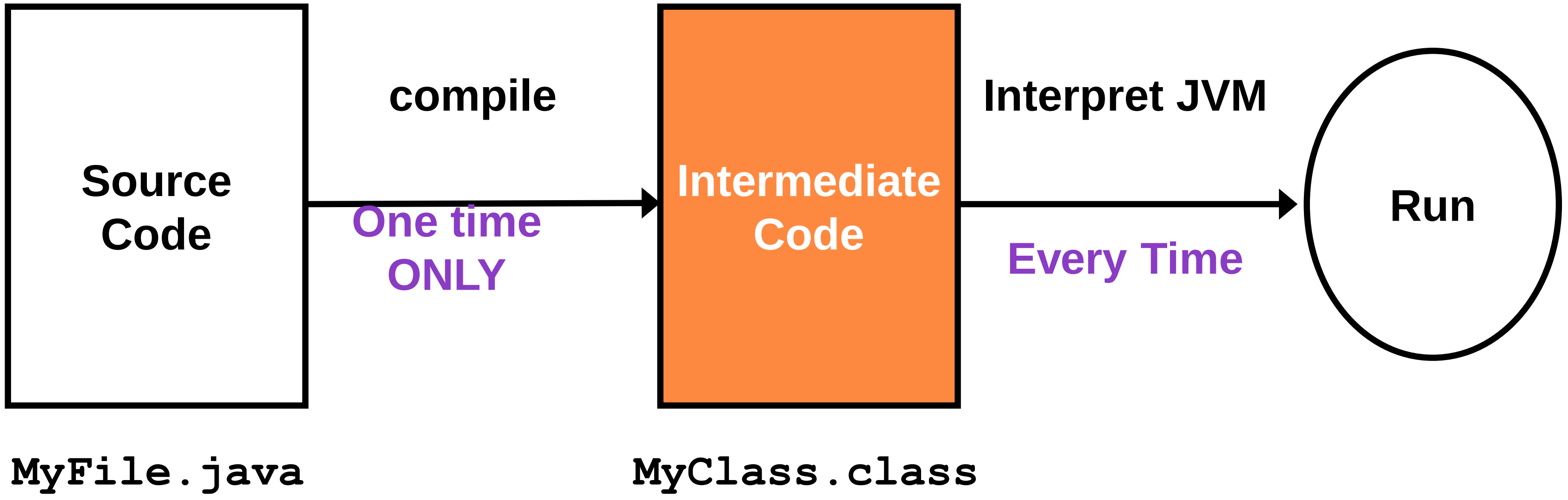
Java Features cont'd

- Machine and Platform Independent



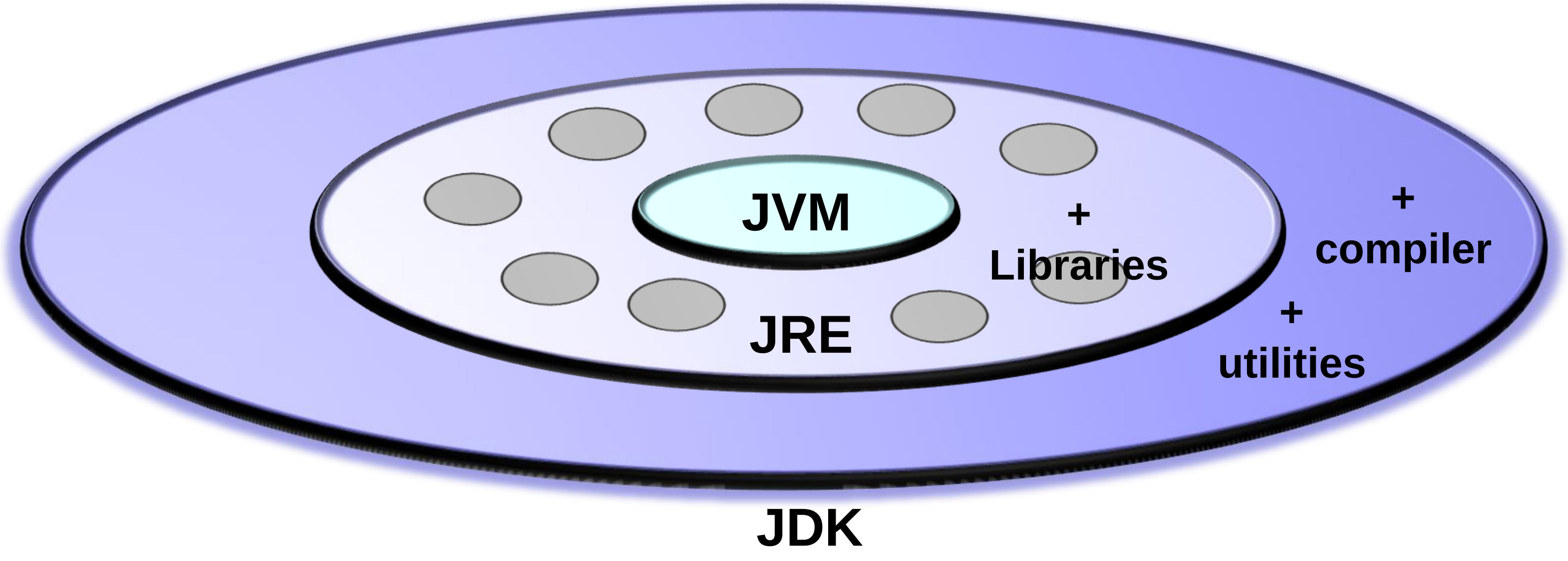
Java Features cont'd

- Java is both, compiled and interpreted



Java Features cont'd

- Java depends on dynamic linking of libraries



Java development Kit (JDK)

Java Features cont'd

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

Java Environment Setup

- Download JDK 8

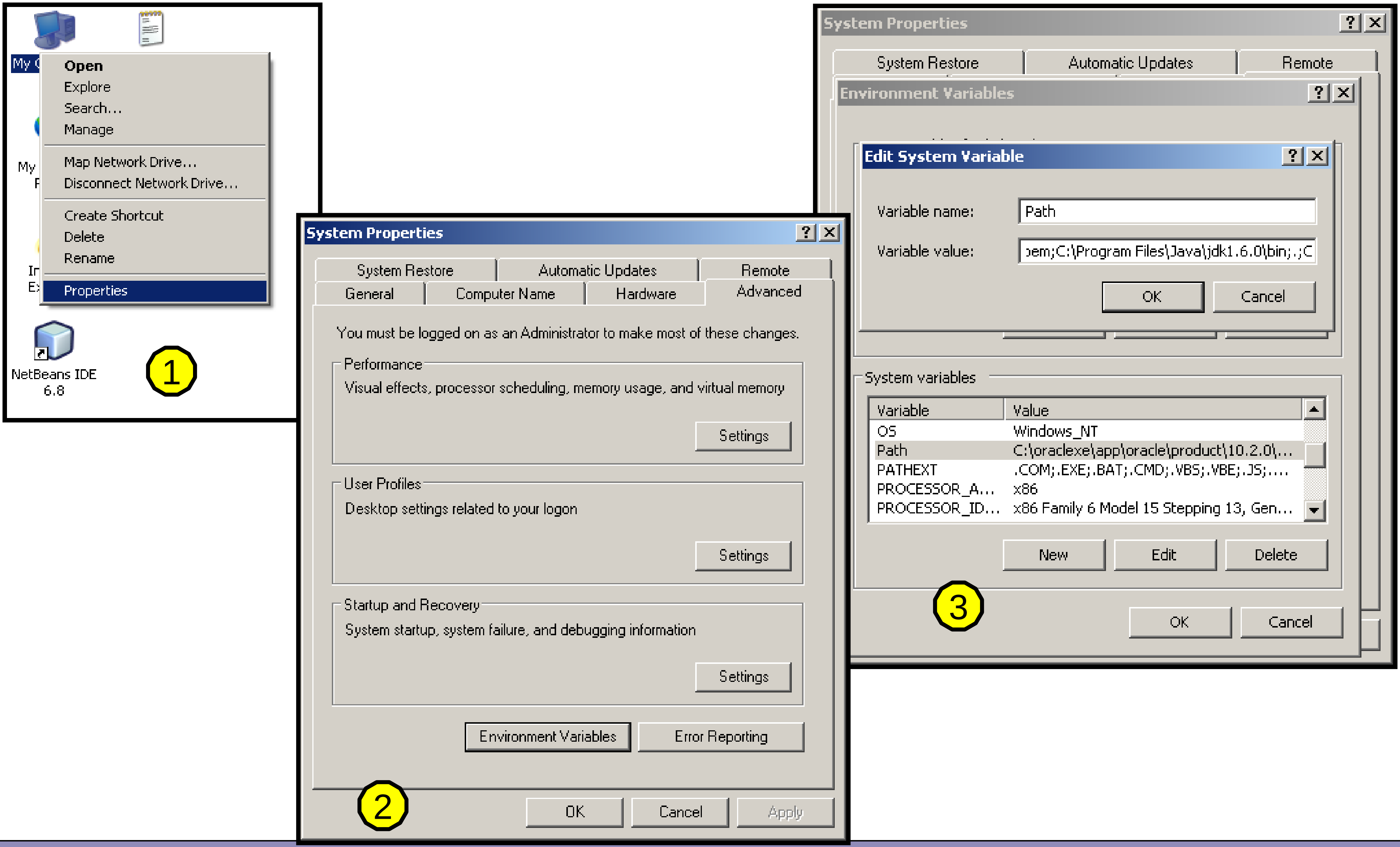
  http://bit.ly/3TJqjJA

Java Environment Setup

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

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

Java Environment Setup



System Properties

System Restore | Automatic Updates | Remote
General | Computer Name | Hardware | Advanced

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings

User Profiles
Desktop settings related to your logon
Settings

Startup and Recovery
System startup, system failure, and debugging information
Settings

Environment Variables | Error Reporting

OK | Cancel | Apply

System Properties

System Restore | Automatic Updates | Remote

Environment Variables

Edit System Variable

Variable name: Path

Variable value: ɔem;C:\Program Files\Java\jdk1.6.0\bin;.;C

OK | Cancel

System variables

Variable	Value
OS	Windows_NT
Path	C:\oraclexe\app\oracle\product\10.2.0\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86
PROCESSOR_ID...	x86 Family 6 Model 15 Stepping 13, Gen...

New | Edit | Delete

OK | Cancel

First Java Application

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

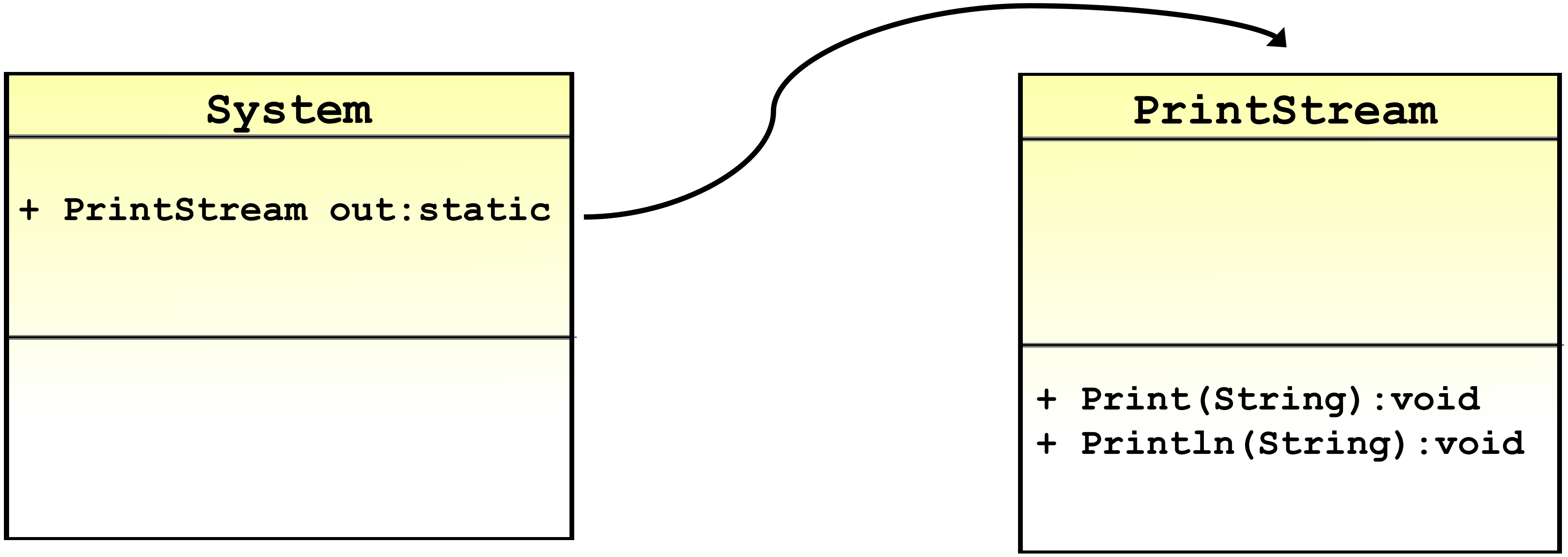
File name: `hello.java`

First Java Application cont’d

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

System.out.println("Hello");

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



Compiling and Running a Java Application

- **To compile:**

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

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

- **To run:**

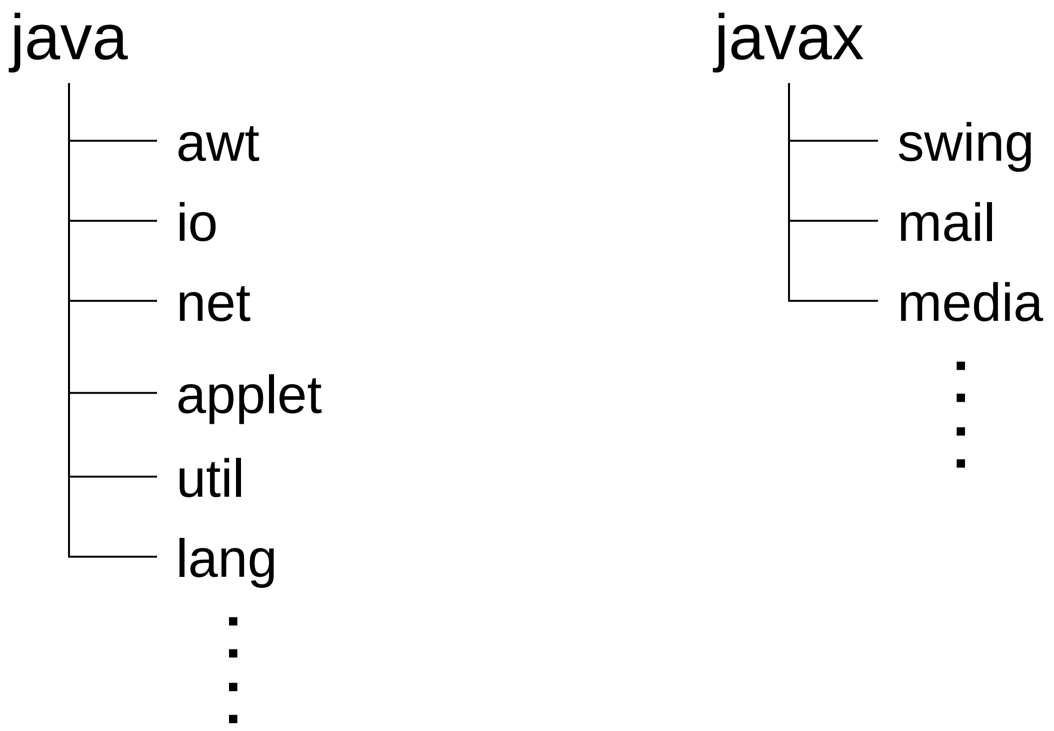
```
Prompt> java HelloWorld
Hello Java
Prompt>
```

Java Structure

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

Java Structure cont’d

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



Specifying a Package

- If no package is specified,
  - then the compiler places the .class file in the default package (i.e. the same folder of the .java file).

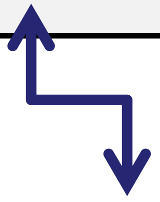
- To specify a package for your application,
  - write the following line of code at the beginning of your class:

```
package mypkg;
```

Specifying a Package

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

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



- To run:

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

JAR File

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

How to make JAR file

- To create a compressed JAR file:

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

- **Example:**

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

How to make JAR file cont'd

- To create an executable JAR file:
  1. Create text file that list the main class.

     "The class that has the main method"
  2. Write inside the text file this text:

     Main-Class: <class name>
  3. Then run the jar utility with this command line:

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

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:

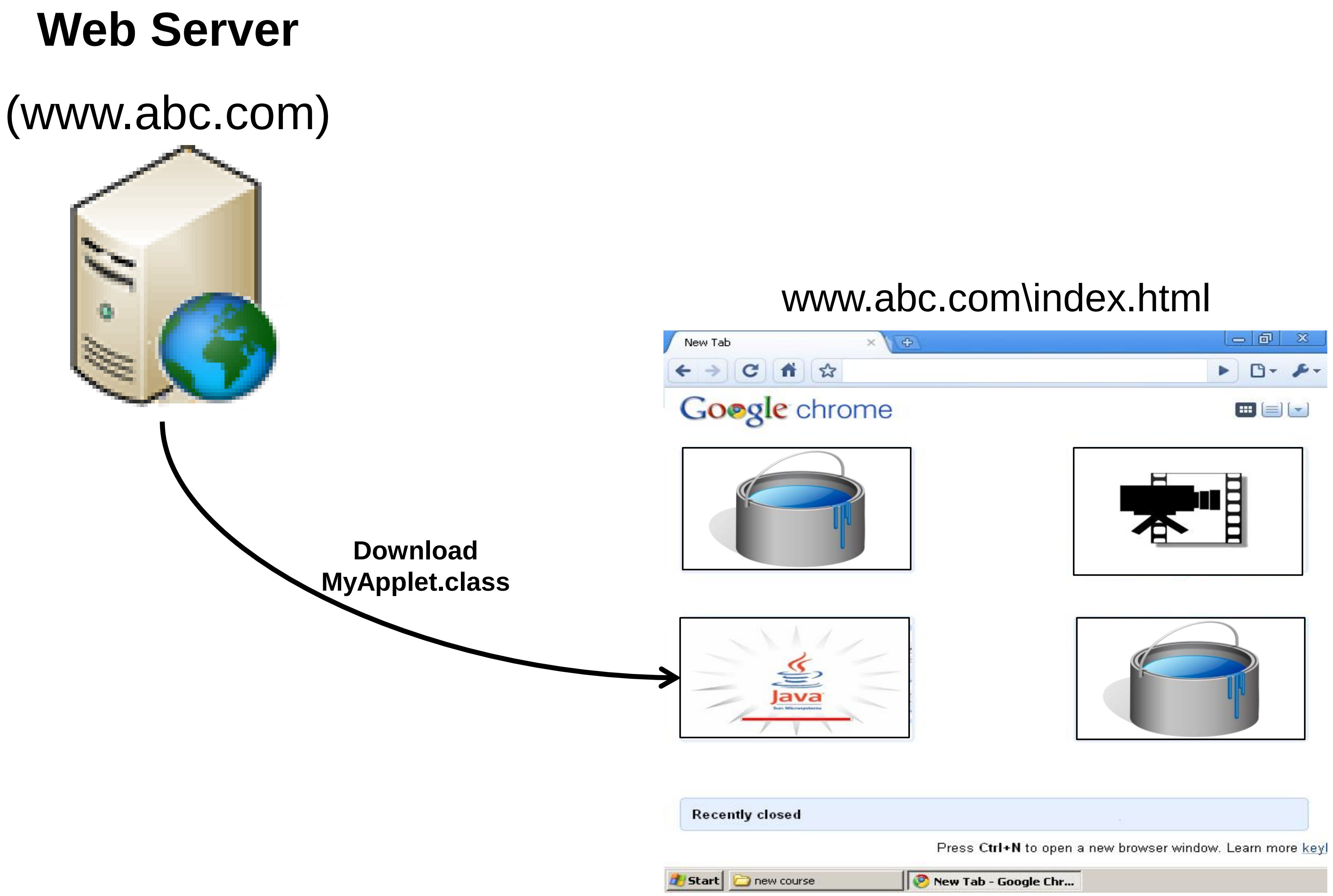
  `MY_STATIC_VAR , MAX_NUMBER`

- Package:

  `pkg1 , util , accesslayer`

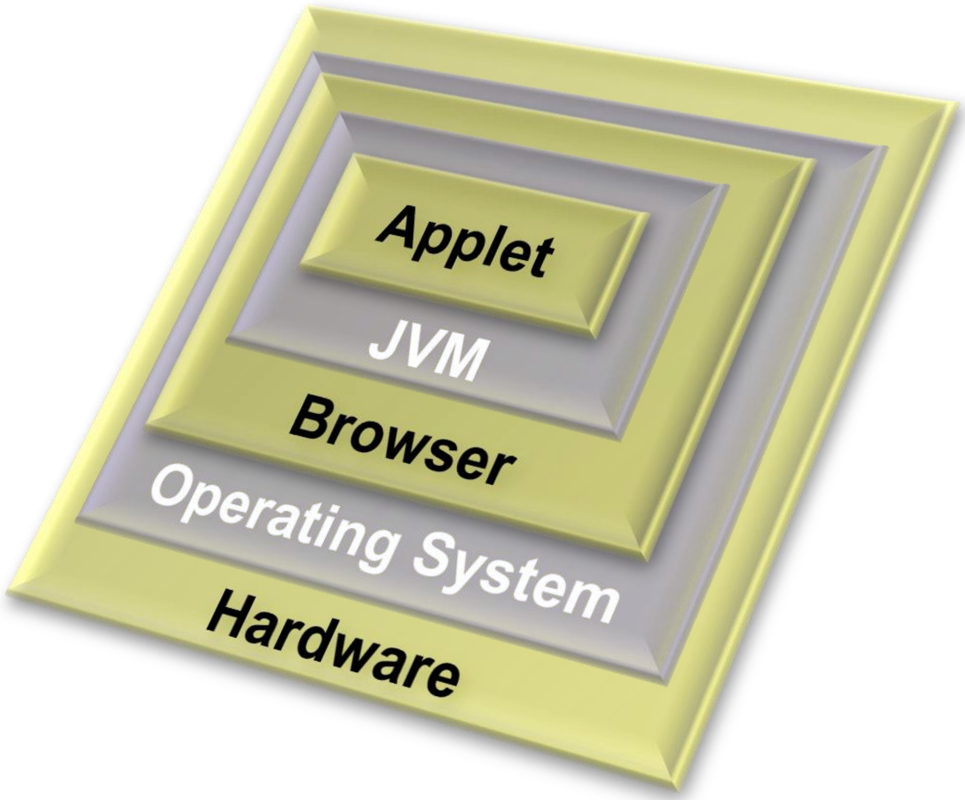
Applets

Overview


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

Applet Features

- Machine and Platform Independent



Applets

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

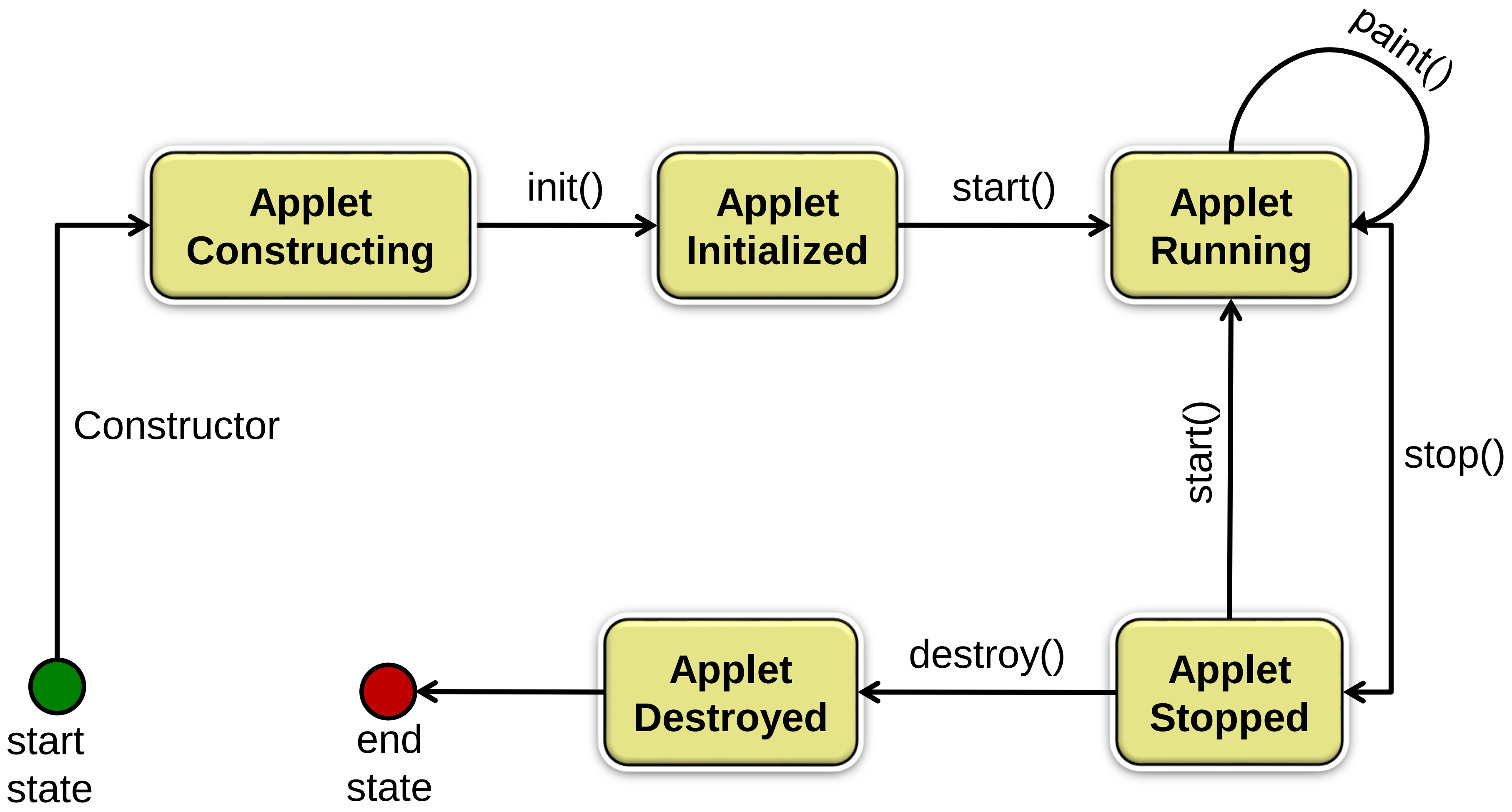
- The .class file of the applet is downloaded from the web server to the client’s machine

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

Applet Security

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

Applet Life Cycle



Applet Life Cycle

- **The life cycle of Applet:**
  - **`init()`**:
    - called when the applet is being initialized for the first time.
  - **`start()`**:
    - called whenever the browser's window is activated.
  - **`paint(Graphics g)`**:
    - called after **`start()`** to paint the applet, or
    - whenever the applet is repainted.
  - **`stop()`**:
    - called whenever the browser's window is deactivated.
  - **`destroy()`**:
    - called when the browser's window is closed.

Applet Life Cycle cont'd

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

Basic Java Applet

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

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

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

Basic Java Applet cont'd

- In order to run the applet we have to create a simple HTML web page, then we invoke the applet using the <applet> tag.

- The **`<applet>`** tag requires 3 mandatory attributes:
  - code
  - width
  - height

- An optional attribute is codebase, which specifies the path of the applet's package.

Basic Java Applet cont'd

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

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

Compiling and Running an Applet

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

- To compile write in cmd this command:

```
javac HelloApplet.java
```

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

```
appletviewer mypage.html
```

from the command prompt.

Lab Exercise

1. 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

Introduction to OOP

- **What is OOP?**
  - OOP is mapping the real world to Software
  - OOP is a ***community*** of interacting agents called ***objects***.
  - Each object has a role to play.
  - 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***.

Introduction to OOP

- **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.

Introduction to OOP - Class

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

Introduction to OOP – Object & Class

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

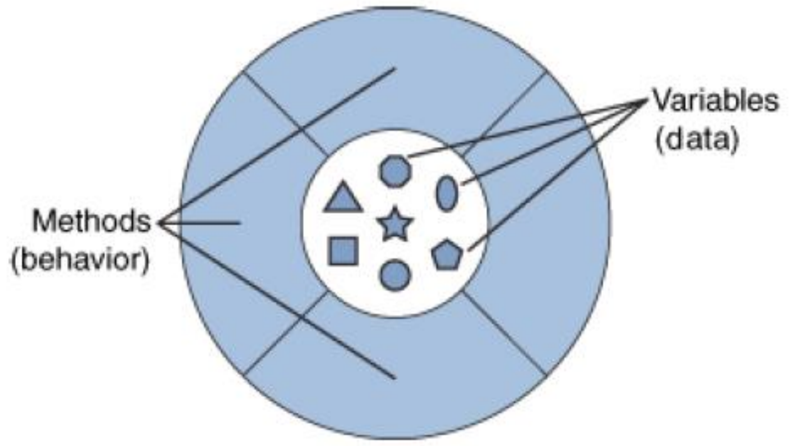
class

object

Introduction to OOP - 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)



Introduction to OOP – Object & Class

- 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.

Introduction to OOP – Object & Class

- 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 SerratedDisc {
size
numberOfPins
spin
}
	SerratedDisc serr_Car =
	new SerratedDisc ();
	
	size = 10
	numberOfPins = 8

How to create a class?

- To define a class, we write:

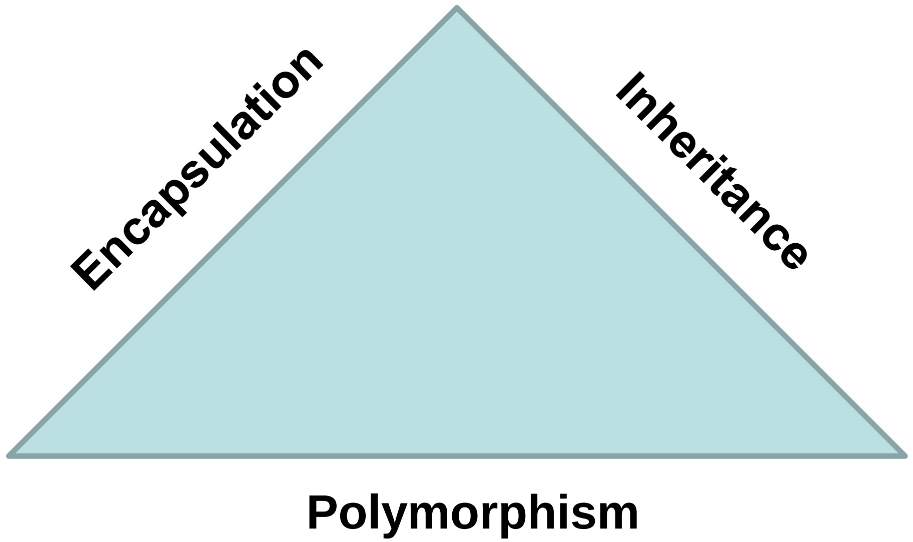
```
<access-modifier>* class <name>
{
        <attributeDeclaration>*
        <methodDeclaration>*
        <constructorDeclaration>*
}
```

- Example:

```java
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:



Encapsulation

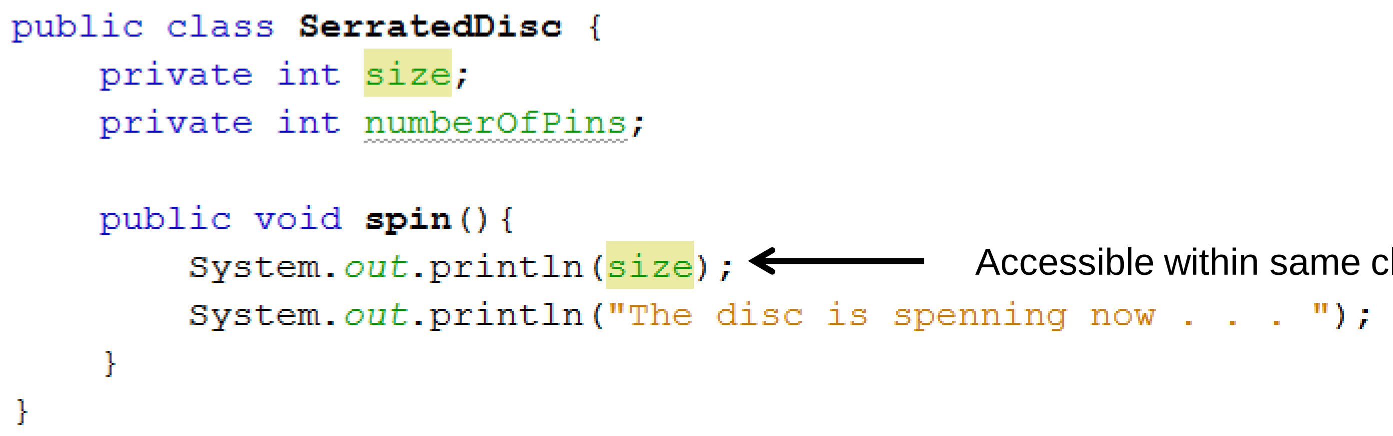
- It is to encapsulate the data and behaviors in one class.
- In addition it is a language mechanism for restricting access to some of the object's components.
- Data hiding is done in Java using the public, and private key words.

```
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

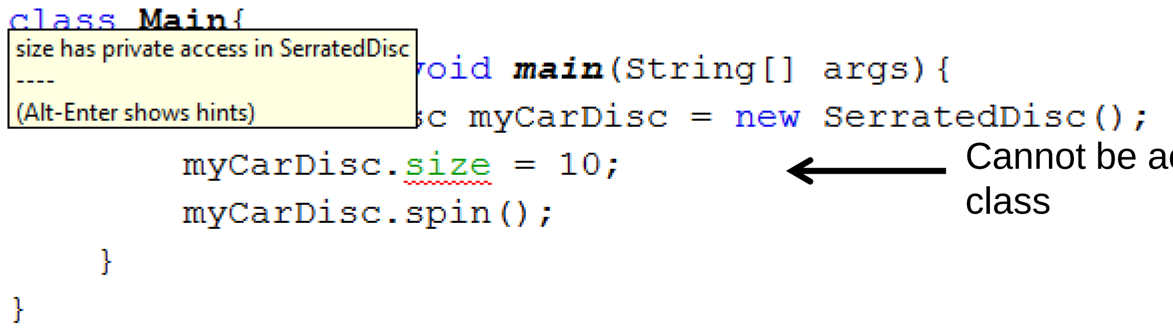
Public class members can be accessed within the class or from another class

Encapsulation cont'd



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

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

Accessible within same class



```
class Main{
              oid main(String[] args){
              c myCarDisc = new SerratedDisc();
        myCarDisc.size = 10;
        myCarDisc.spin();
    }
}
```

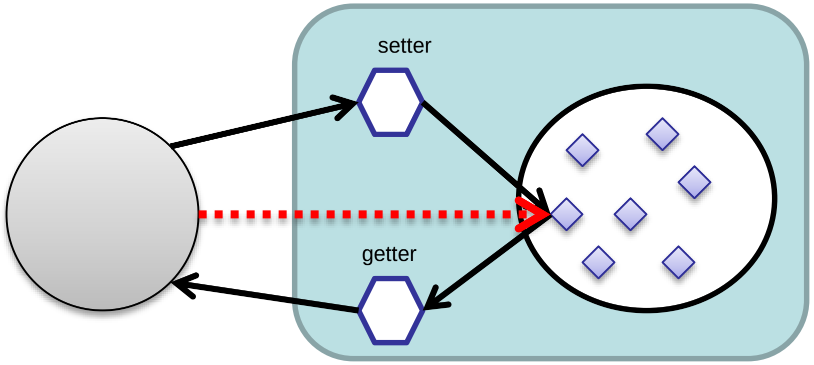
size has private access in SerratedDisc

(Alt-Enter shows hints)

Cannot be accessed outside the class

Encapsulation cont'd

- 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.



Encapsulation cont’d

- 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;
}
```

Encapsulation cont’d

- 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();
    }
}
```

Coding Guidelines

- Think of an appropriate name for your class.
  - Don't use XYZ or any random names.

- Class names starts with a CAPITAL letter.
  - not a requirement it is a convention

Declaring Methods

- To declare a method:

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

- Example:

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

Declaring Methods

- The following are characteristics of methods:
  - It can return one or no values
  - It may accept as many parameters it needs or no parameter at all.
  - 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.

Class and constructor

- 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).

Class and constructor cont’d

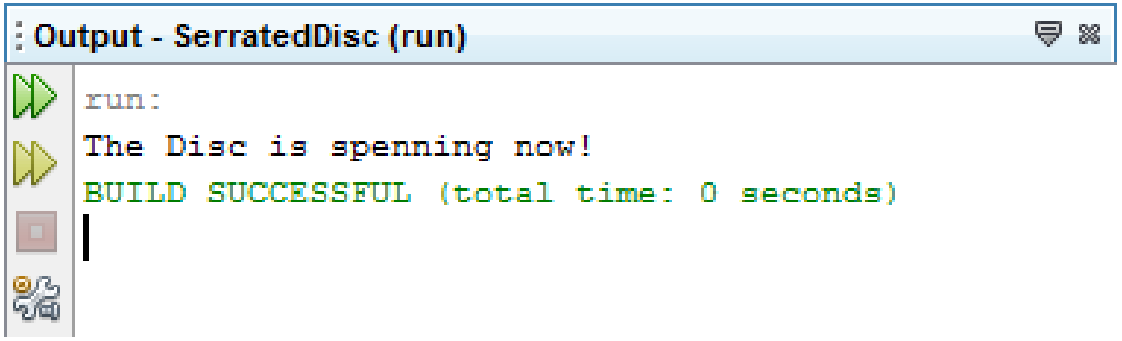
- By default any class - we write - has a constructor without the need to write it, called default constructor.
- The default constructor does nothing but it must be called before creating an object.
- The constructor is the best place to put initialization per-object.
- If no constructors are written 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.

Class and constructor cont'd

- 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



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

Class and constructor cont'd

- 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;
    }
}
```

Class and constructor cont'd

- 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.

Instance and Static Members Members

- 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

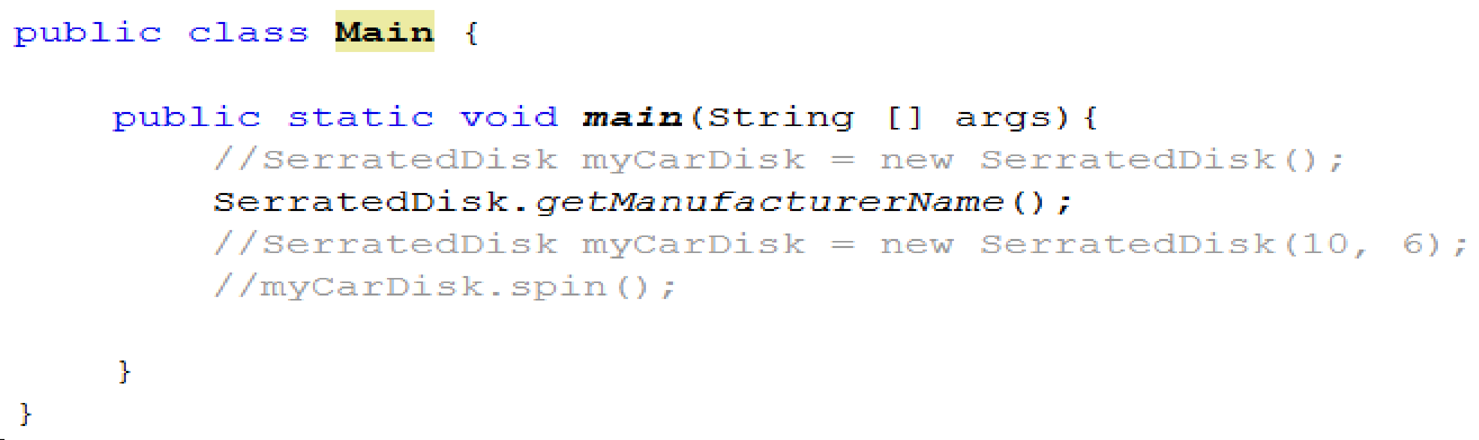
- There is some cases we must use the this reference, like:

```
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.

Instance and Static Members cont'd

- 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

```java
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

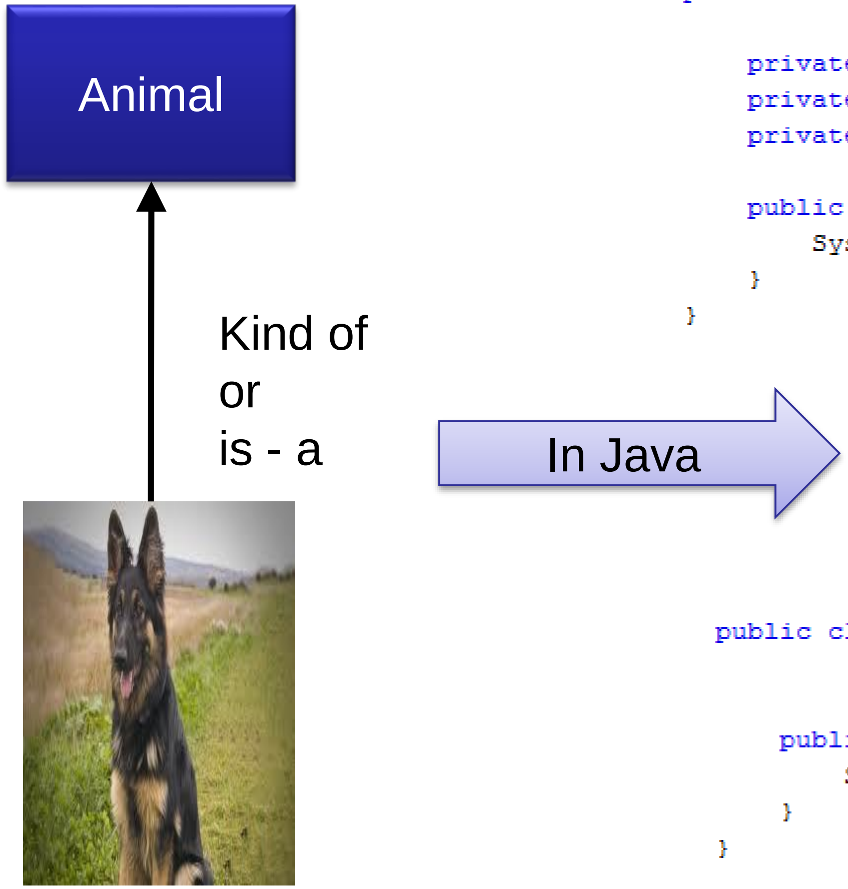
OOP II and Applet

Inheritance

- **Inheritance:**
  - 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



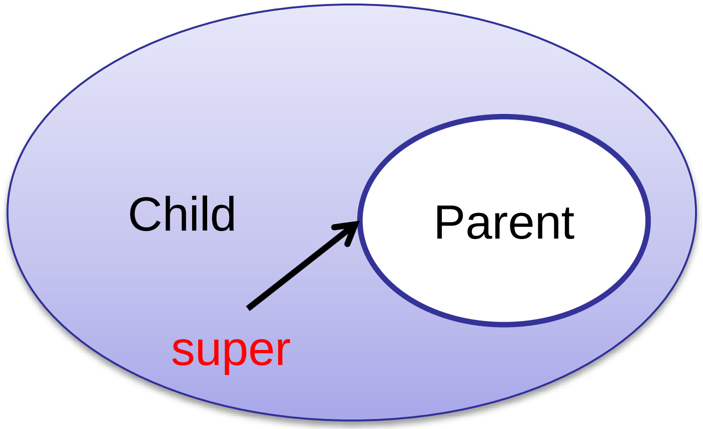
```
public class Animal {

    private String name;
    private int numOfLegs;
    private float speed;

    public void play(){
        System.out.println("the animal is playing......");
    }
}
```

```
public class Dog extends Animal{


    public void woof(){
        System.out.println("Dog: Woof Woof....");
    }
}
```

Inheritance

Inheritance

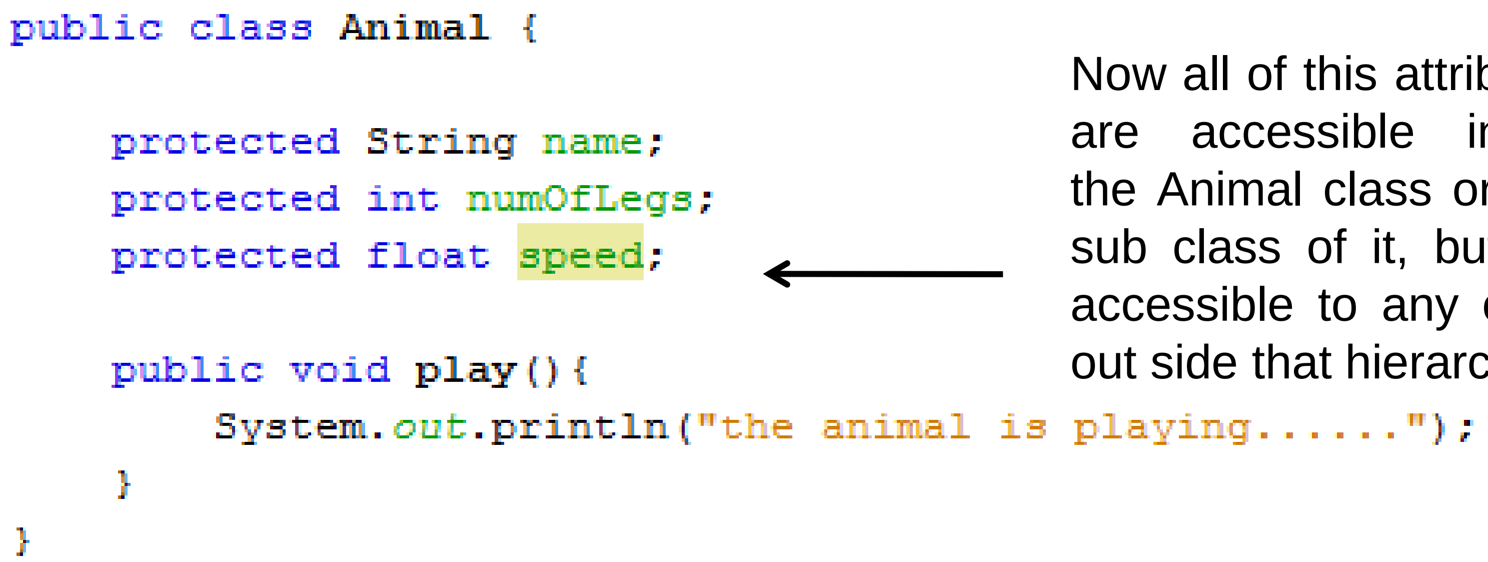
- the super reference can only point to parent public members.



- 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.

Inheritance

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



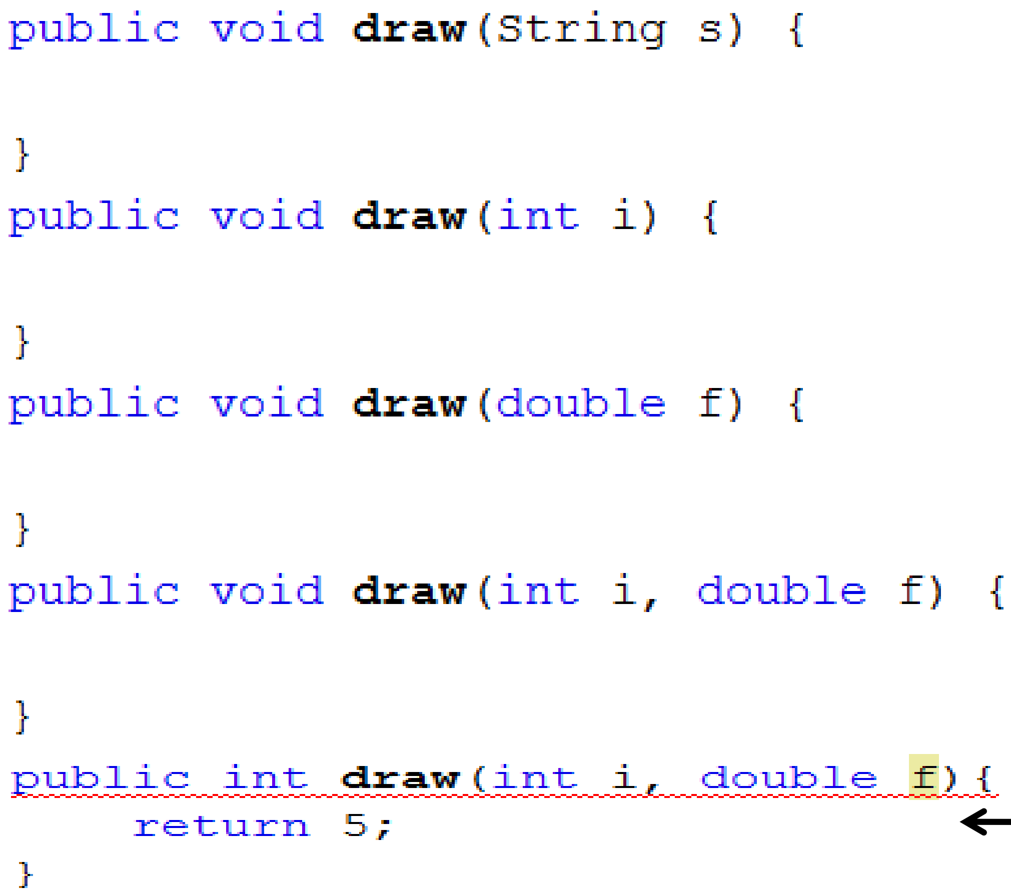
```
public class Animal {

    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 out side that hierarchy

polymorphism

- 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)



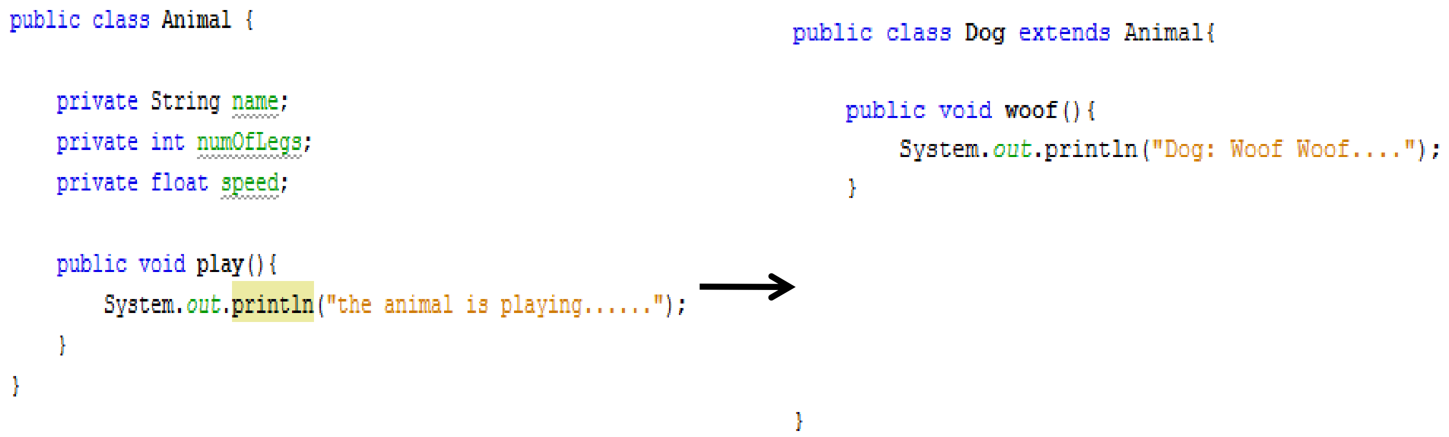
```
public void draw(String s) {

}
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 {

    private String name;
    private int numOfLegs;
    private float speed;

    public void play(){
        System.out.println("the animal is playing......");
    }
}
```

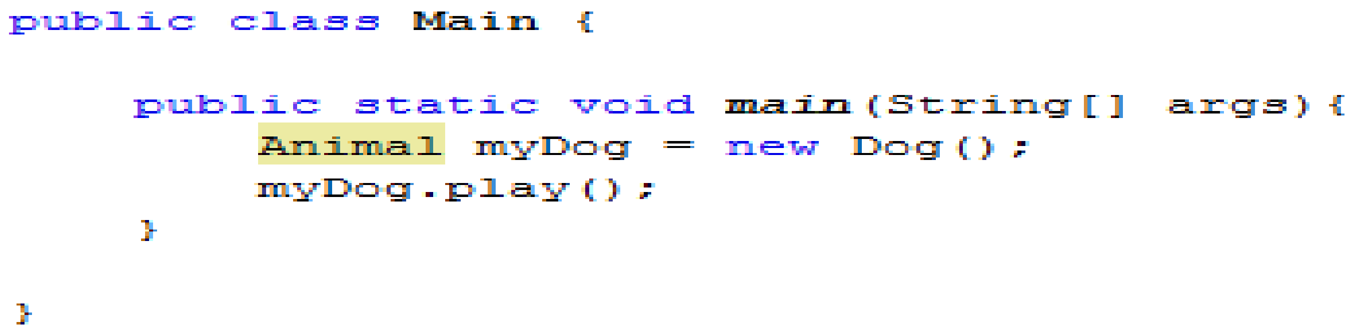
```
public class Dog extends Animal{

    public void woof(){
        System.out.println("Dog: Woof Woof....");
    }


}
```

Polymorphism (overriding)

- 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

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

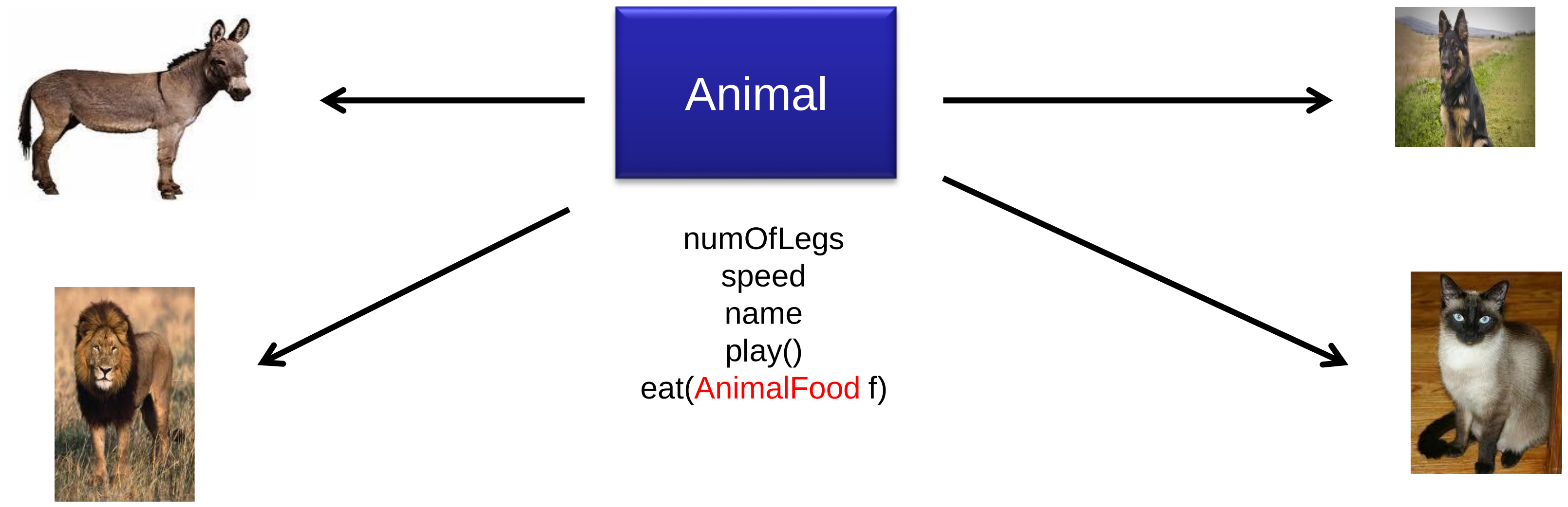


```
public class Main {

    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.

Data Abstraction

- 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.



numOfLegs
speed
name
play()
eat(AnimalFood f)

Data Abstraction

- At this point the class Animal is not will 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 this properties but it is only abstract definition.
- So its 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.

Data Abstraction

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

```
public abstract class Animal {
```

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

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

Data Abstraction

- 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

Identifiers

- An identifier is the name given to a feature (variable, method, or class).

- An identifier can begin with either:
  - a letter,
  - $, or
  - underscore.

- Subsequent characters may be:
  - a letter,
  - $,
  - underscore, or
  - digits.

Data types

- Data types can be classified into two types:

Primitive

Reference

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

Arrays
Classes
Interfaces

Wrapper Classes

- Each primitive data type has a corresponding wrapper class.

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

Wrapper Classes cont'd

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

  1. As an argument of a method that expects an object.
  2. To use constants defined by the class,
     - such as **MIN_VALUE** and **MAX_VALUE**,

       that provide the upper and lower bounds of the data type.
  3. To use class methods for
     - converting values to and from other primitive types,
     - converting to and from strings,
     - converting between number systems (decimal, octal, hexadecimal, binary).

Wrapper Classes cont’d

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

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

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

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

Wrapper Classes cont'd

- They have special static representations, for example:

POSITIVE_INFINITY		In class Float & Double
NEGATIVE_INFINITY		
NaN	Not a Number	

Literals

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

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

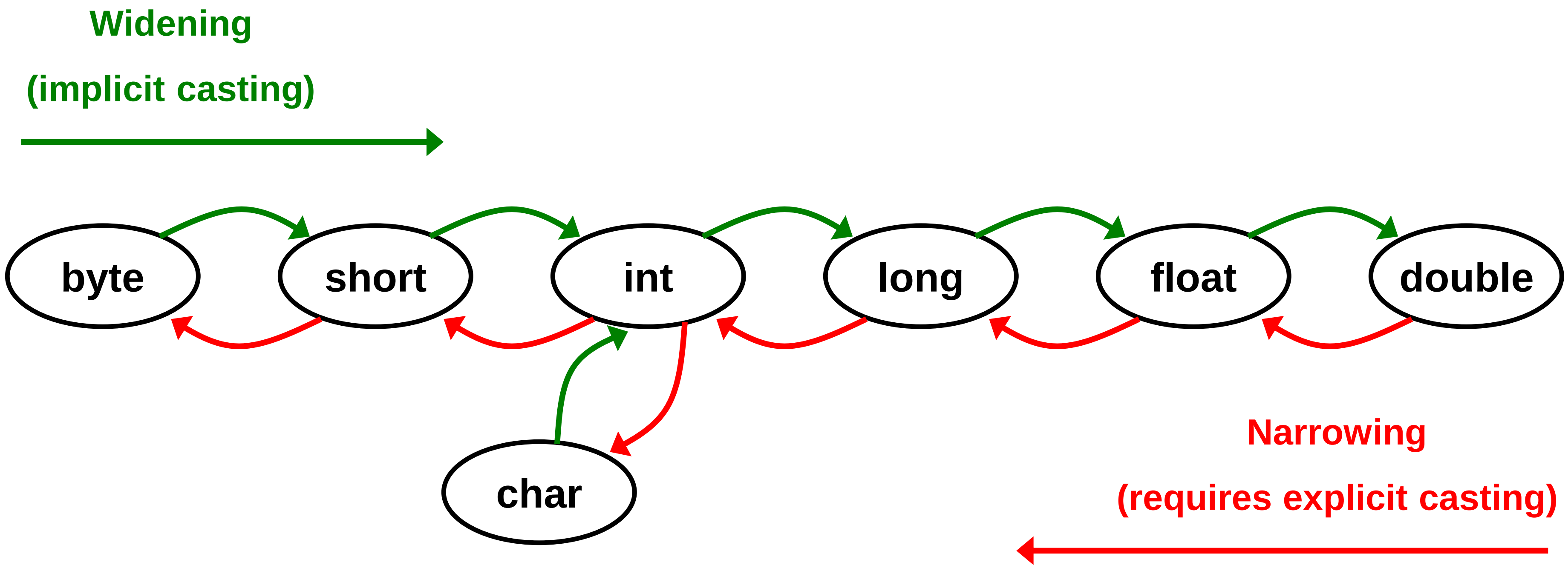
Operators

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

Operators cont'd

- Unary Operators:

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



Operators cont'd

- Arithmetic Operators:

+	-	*	/	%
add	subtract	multiply	division	modulo

- Assignment Operators:

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

- Relational Operators:

<	<=	>	>=	==	!=	instanceof

Operations must be performed on homogeneous data types

Operators cont’d

- Shift Operators:

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

- Bitwise and Logical Operators:

&	|	^
AND	OR	XOR

- Short Circuit Operators:

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

Operators cont'd

- Ternary Operator:

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

```
int y=15;
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=

Reference Data types: Classes

- General syntax for creating an object:

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

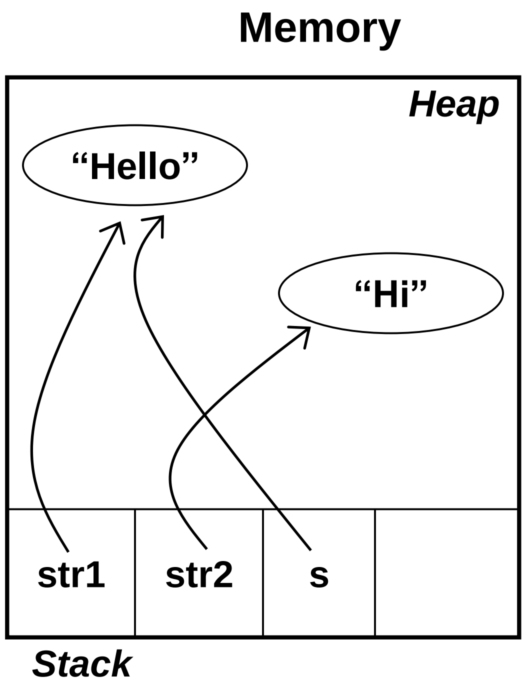
- Or on one line:

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

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

Reference Data types: Classes cont'd

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

String str2 = new String("Hi");

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

str1 = null;

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



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



Lesson 5

Using Arrays & Strings

What is Array?

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

What is Array?

- Java Arrays are homogeneous.

- You can create:
  - An array of primitives,
  - An array of object references, or
  - An array of arrays.

- If you create an array of object references, then you can store subtypes of the declared type.

Declaring an Array

- General syntax for creating an array:

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

- Or on one line, hard coded values:

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

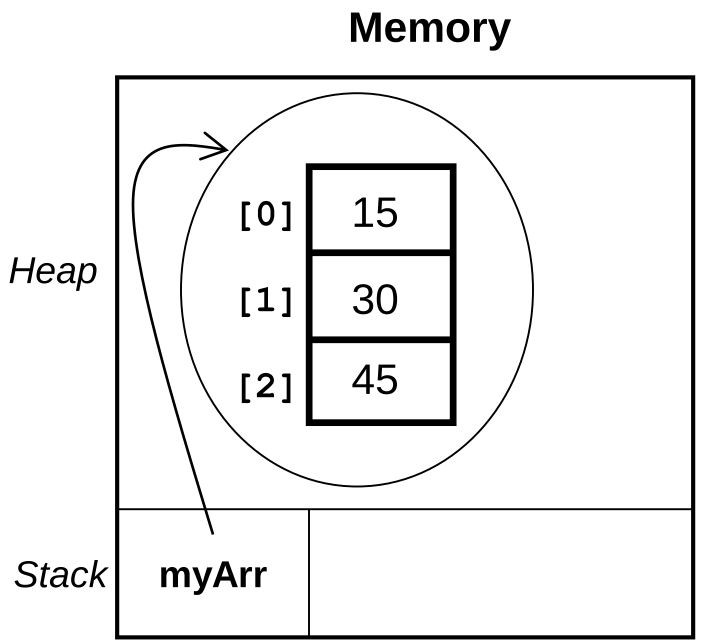
- To determine the size (number of elements) of an array at runtime, use:

```
arrayIdentifier.length
```

Declaring an Array cont'd

- **Example1:** Array of Primitives:

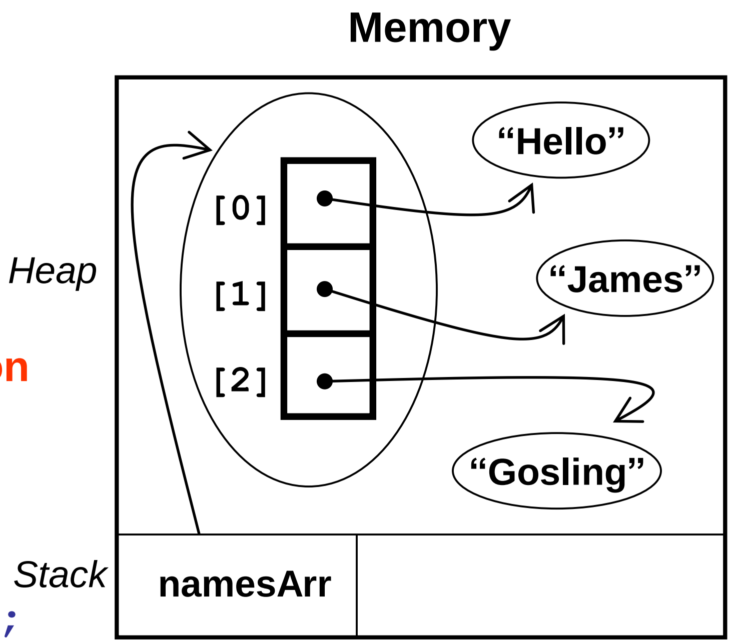
```
int[] myArr;

myArr = new int[3];

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

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



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



Declaring an Array cont'd

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

```
String[] namesArr;

namesArr = new String[3];

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

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

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



String Operations

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

  - Creating String Object:

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

  - Testing for String equality:

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

if(myStr1.equalsIgnoreCase(sp1))

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

Strings Operations cont’d

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

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

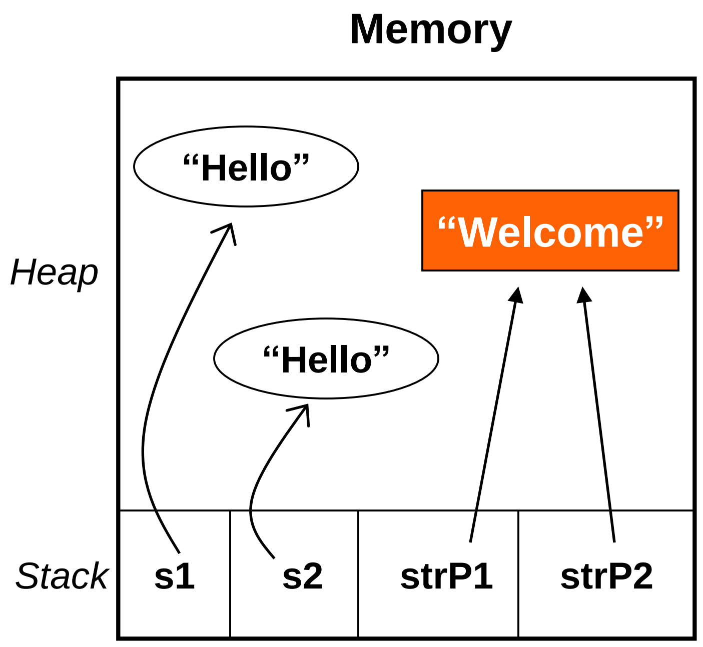
- Objects of class String are immutable
  - you can’t modify the contents of a String object after construction.

- Concatenation Operations always return a new String object that holds the result of the concatenation. The original objects remain unchanged.

String Pool

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

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

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



String Pool cont'd

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

Lesson 6

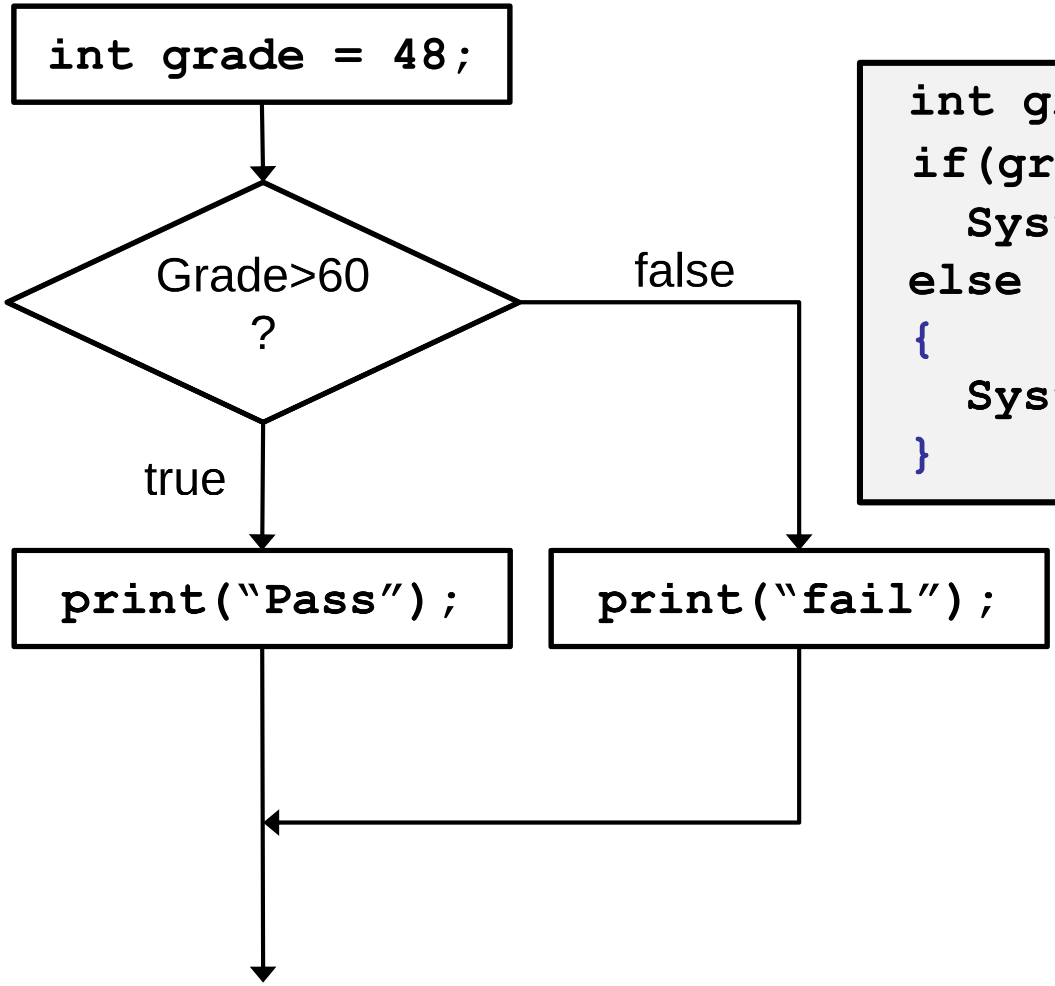
Controlling Program Flow

Flow Control: Branching - if, else

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

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

if, else Example



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

Flow Control: Branching - switch

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

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

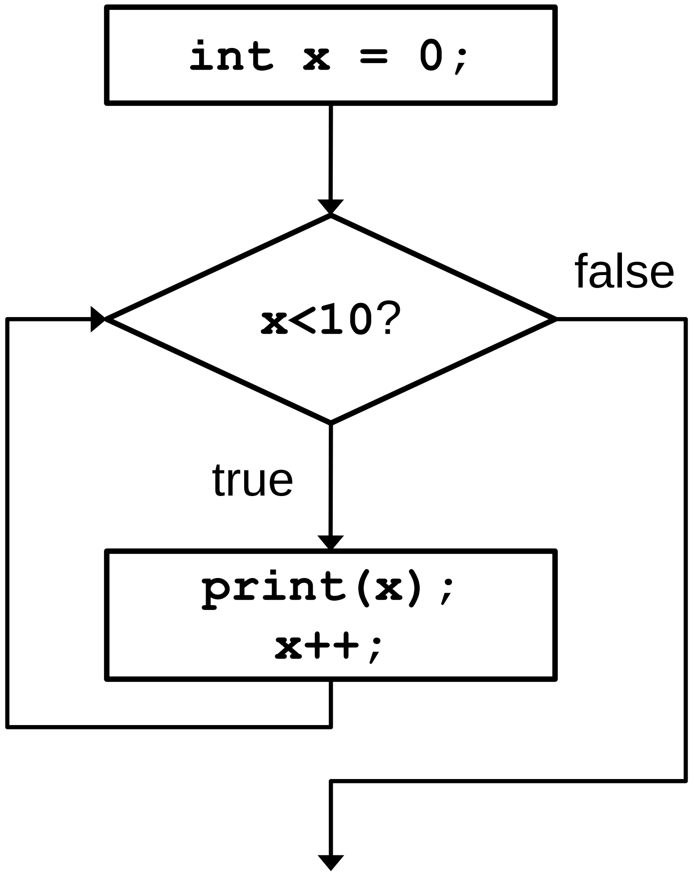
- byte
- short
- int
- char
- enum
- String "Java 7"

Flow Control: Iteration – while loop

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

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

while loop Example



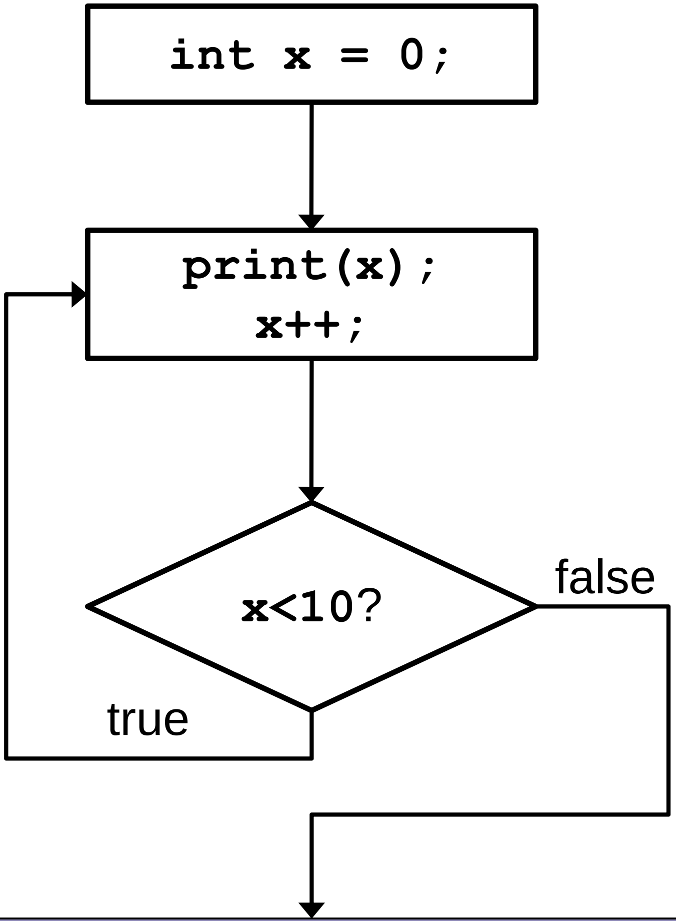
```
int x = 0;
while (x<10) {
    System.out.println(x);
    x++;
}
```

Flow Control: Iteration – do..while loop

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

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

do..while loop Example



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

Flow Control: Iteration – for loop

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

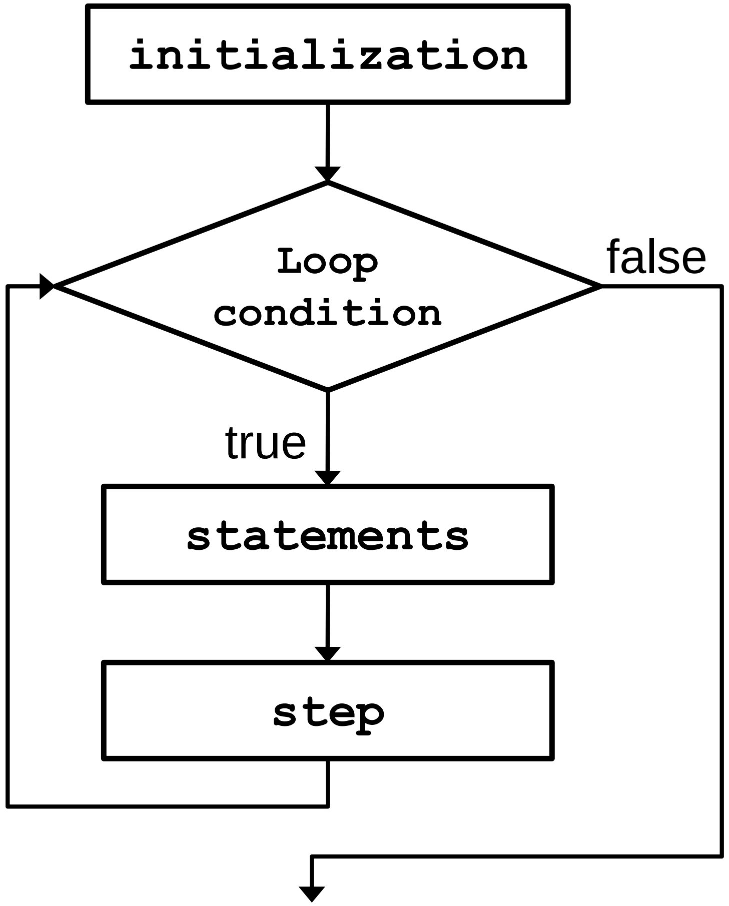
```
for (initialization ; loop_condition ; step)
{
        …
        …
        …
}
```

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

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

Flow Control: Iteration – for loop

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



Flow Control: Iteration –Enhanced for loop

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

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

Flow Control: Iteration –Enhanced for loop example

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

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

average /= samples.length;
```

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

The break statement

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

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

The continue statement

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

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

Comments in Java

- To comment a single line:

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

- To comment multiple lines:

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

Printable Class Documentation (Javadoc)

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

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

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

```
javadoc myfile.java
```

Printable Class Documentation (Javadoc)

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

Example of Javadoc

- Example 1:

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

- Example 2:

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

Lab Exercise

1. Command Line Calculator

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

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

  java Calc 70 + 30

2. String Separator

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

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

  ```
  java IPCutter 163.121.12.30
  ```

- The output should then be:

  ```
  163
  121
  12
  30
  ```

- Write a program that print the following patterns:

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

Lesson 6

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

Lesson 6 Outline

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

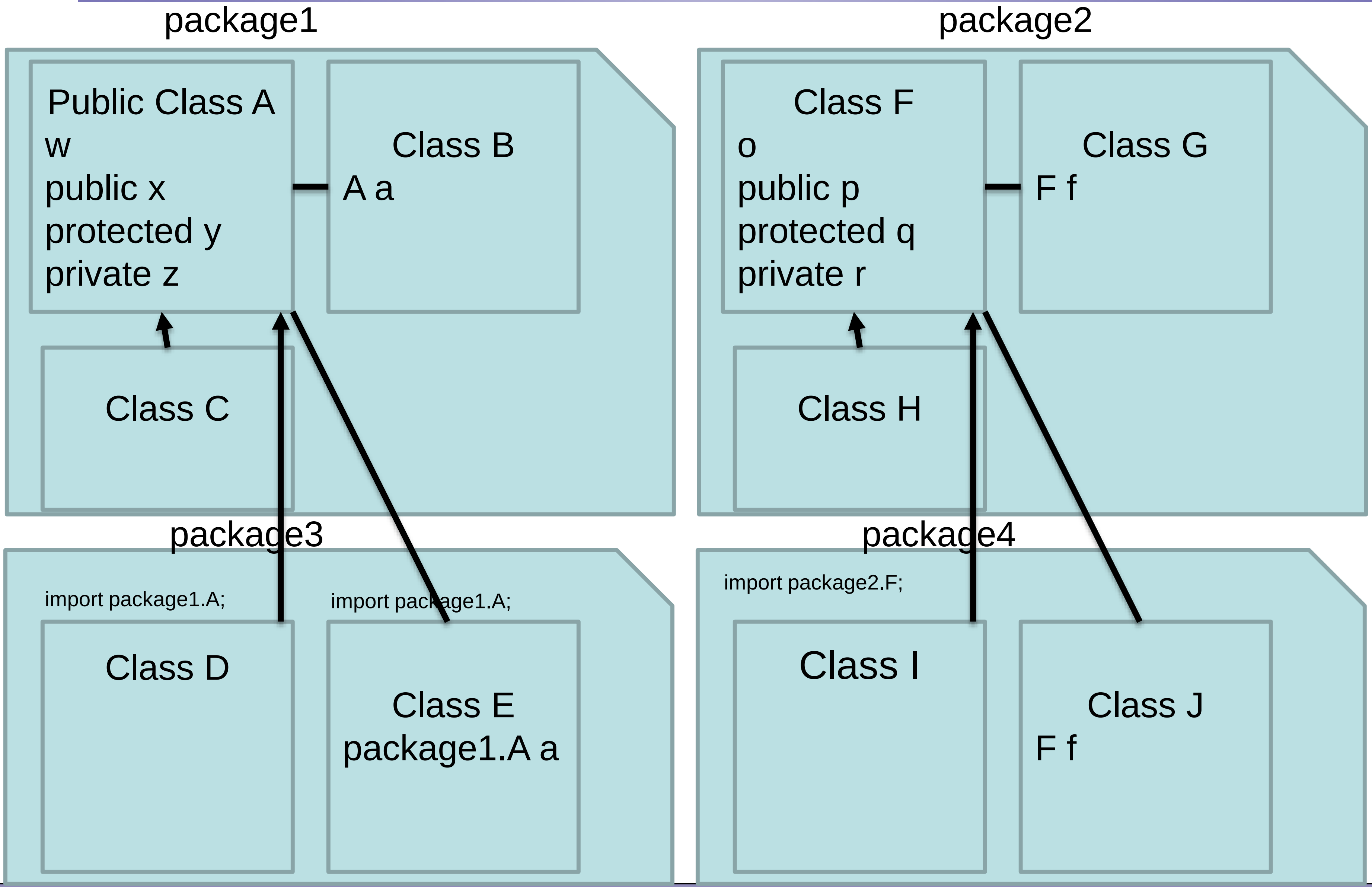
Modifiers and Access Specifiers

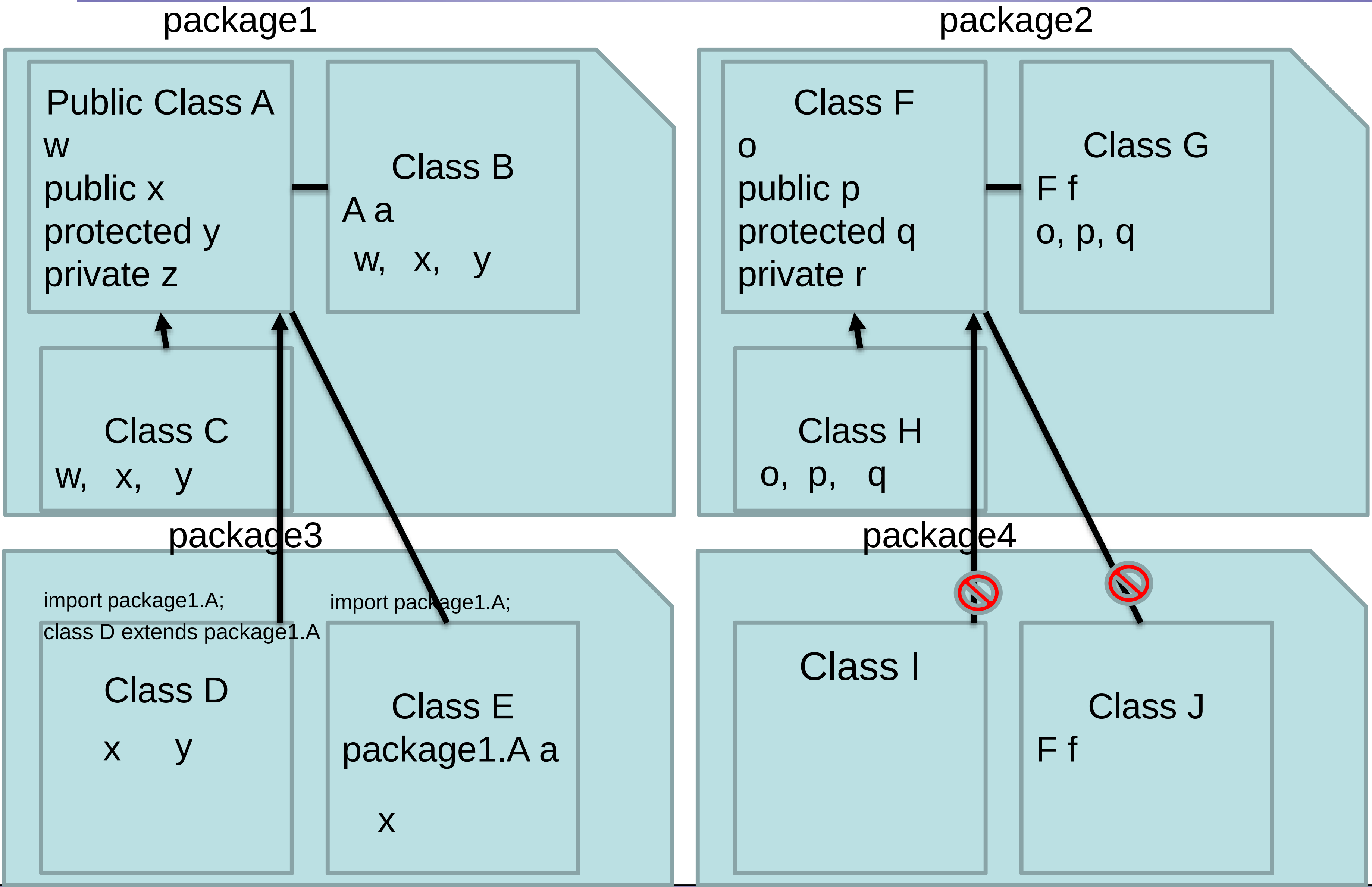
- Modifiers and Access Specifiers are a set of keywords that affect the way we work with features (classes, methods, and variables).

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

Modifiers and Access Specifiers cont'd

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


package1
Public Class A
w
public x
protected y
private z
Class B
A a
Class C
package2
Class F
o
public p
protected q
private r
Class G
F f
Class H
package3
import package1.A;
import package1.A;
Class D
Class E
package1.A a
package4
import package2.F;
Class I
Class J
F f


package1
Public Class A
w
public x
protected y
private z
Class B
A a
w, x, y
Class C
w, x, y
package2
Class F
o
public p
protected q
private r
Class G
F f
o, p, q
Class H
o, p, q
package3
import package1.A;
class D extends package1.A
Class D
x y
import package1.A;
Class E
package1.A a
x
package4
Class I
Class J
F f

Graphics Class

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

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

Graphics Class cont’d

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

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

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

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

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

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

Color Class

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

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

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

- Objects of class Color are immutable.

Color Class cont'd

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

Color Class cont'd

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

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

Font Class

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

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

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

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

Font Class cont’d

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

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

- Objects of class Font are immutable.

Exceptions

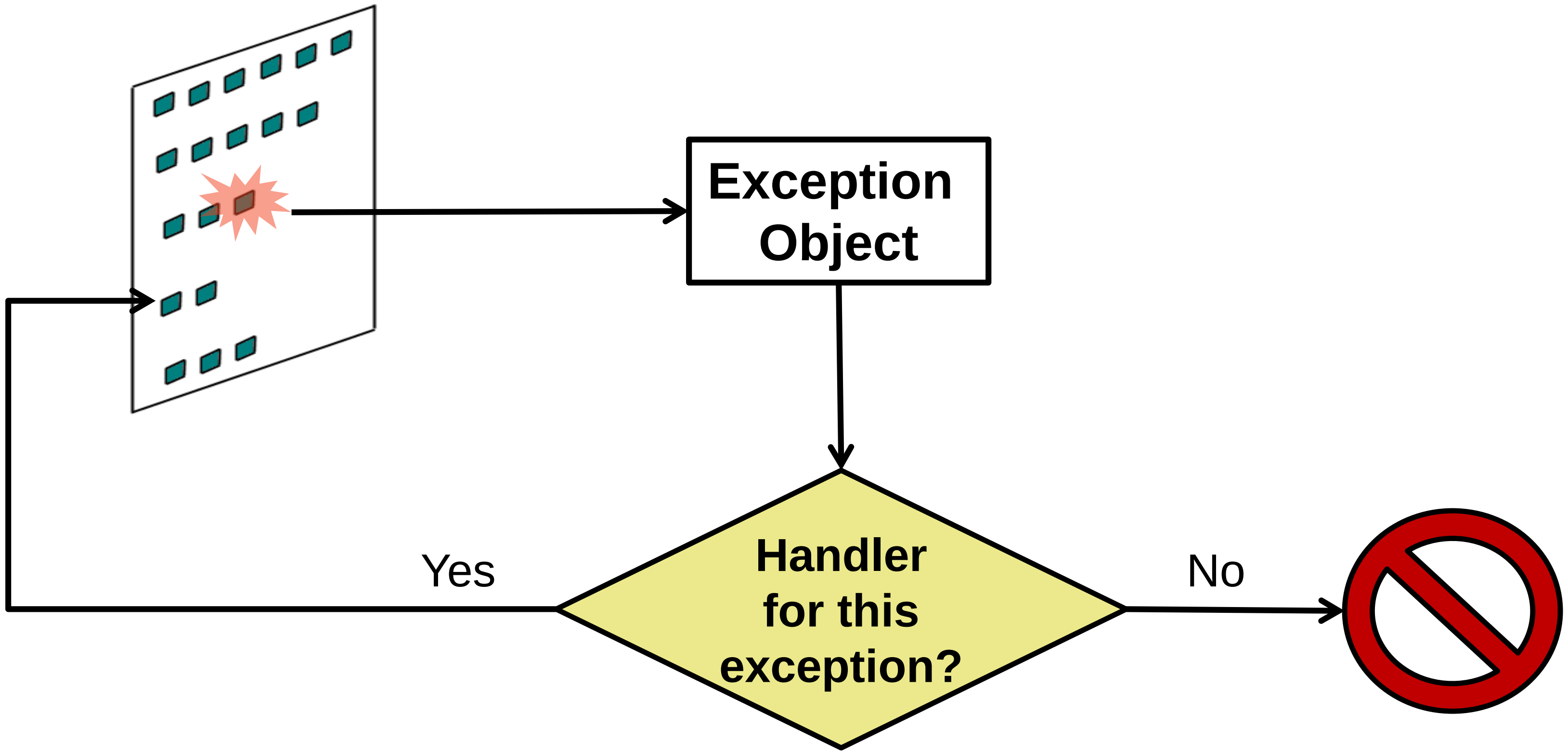
- An exception is an object that's created when an abnormal situation arises in your program **during runtime.**

- **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.

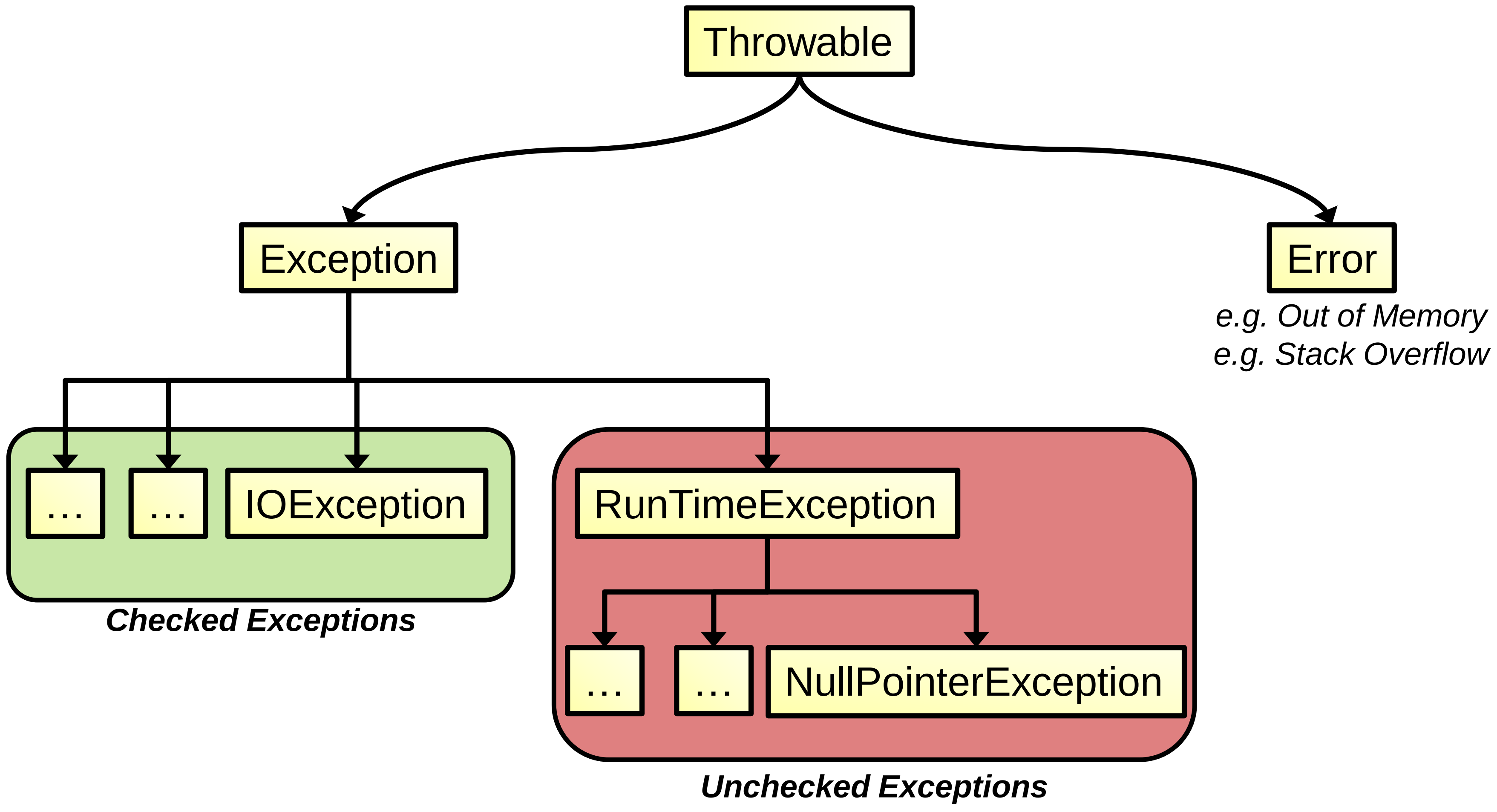
Exceptions cont’d

- 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



Exception Handling

- 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.

Handling Exceptions

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
   }
 }
}
```

Handling Exceptions

- 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
   }
 }
}
```

Throwing Exceptions

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;
  }
}
```

Considerations Regarding Exceptions

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

1. Write separate **`try-catch`** blocks for each method.

2. Put them all inside the same **`try`** block and then write multiple **`catch`** blocks for it
   (one **`catch`** for each exception type).

3. Put them all inside the same **`try`** block and then just **`catch`** the parent of all exceptions: **`Exception`**.

Considerations Regarding Exceptions

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

Example

```
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("....................");
}
```

Considerations Regarding Exceptions cont'd

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

The try-with-resources "Java 7"

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

The try-with-resources example "Java 7"

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

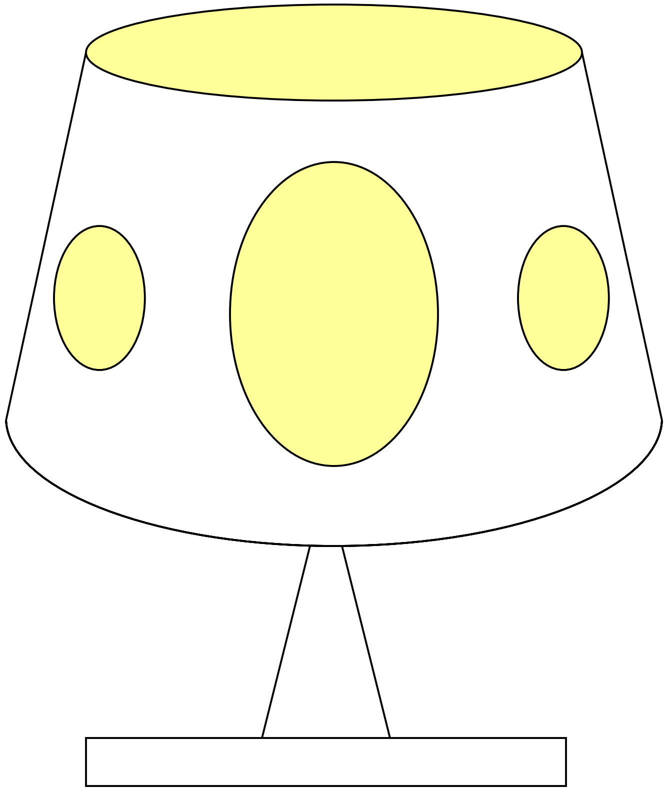
Lab Exercise

1. 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

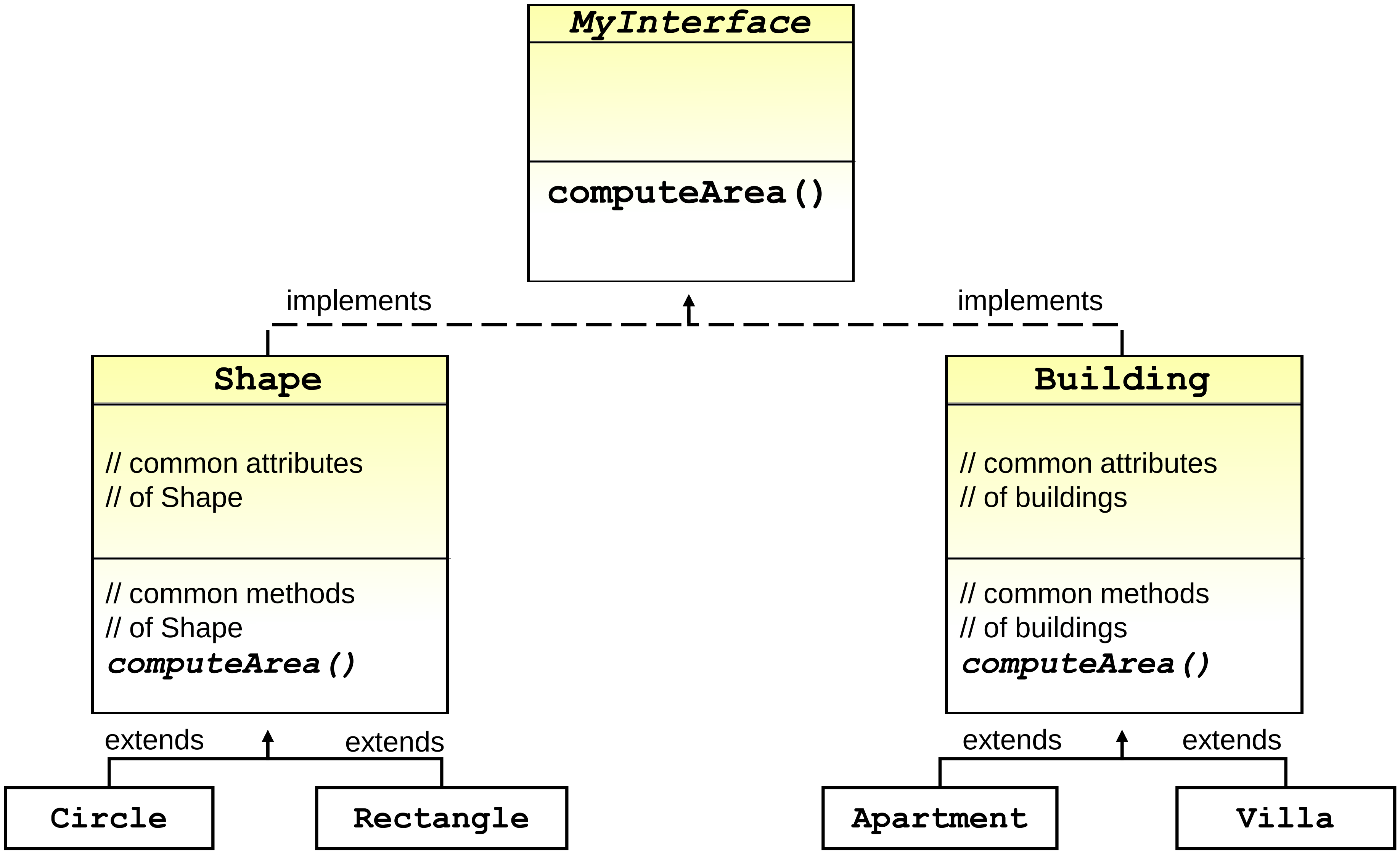
Interfaces

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

Interfaces

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

Example of Interfaces



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.

Interfaces

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

Using interface reference

```
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());
            }
   } }
```

General Consideration about interfaces

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

Lesson 8

Multi-Threading

What is Thread?

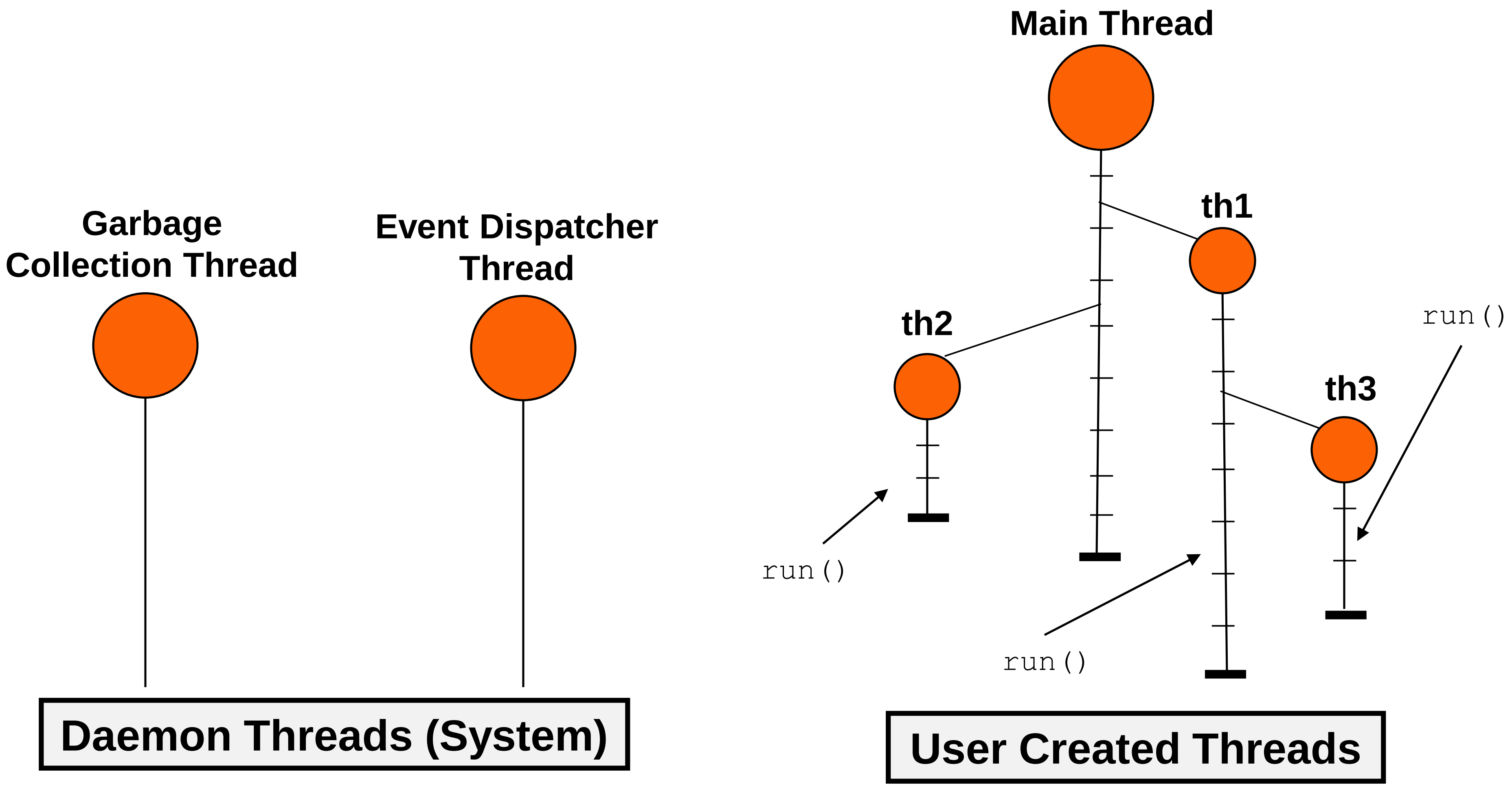
- A Thread is
  - A single sequential execution path in a program
  - Used when we need to execute two or more program segments concurrently (multithreading).
  - Used in many applications:
    - Games , animation , perform I/O
  - Every program has at least two threads.
  - Each thread has its own stack,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

What is Thread?

- 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)

Threads

- several thread objects that are executing concurrently:



Threads cont'd

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

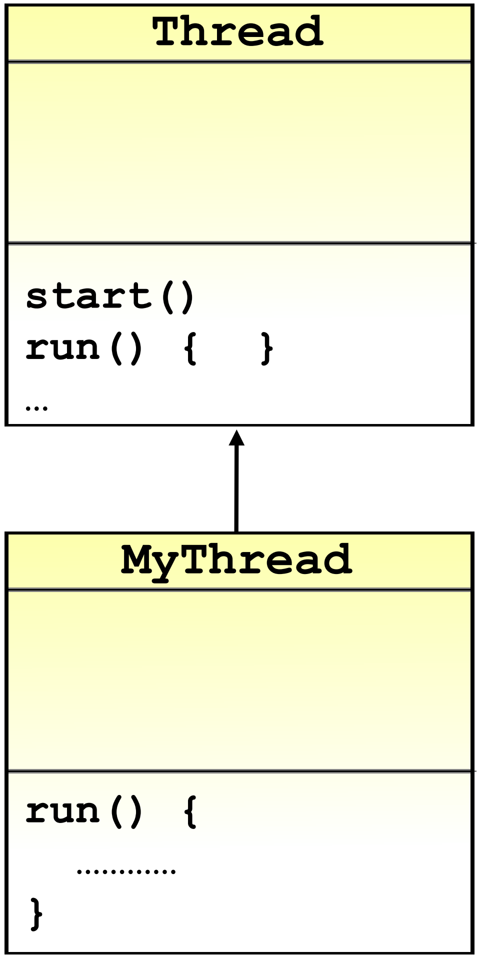
The Thread Class

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

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

Working with Threads

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

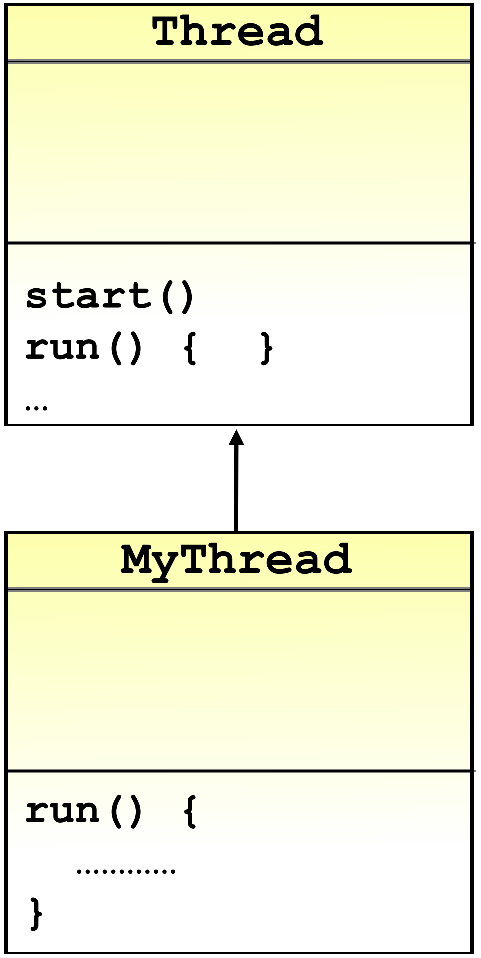


Working with Threads cont'd

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

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

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



Working with Threads

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

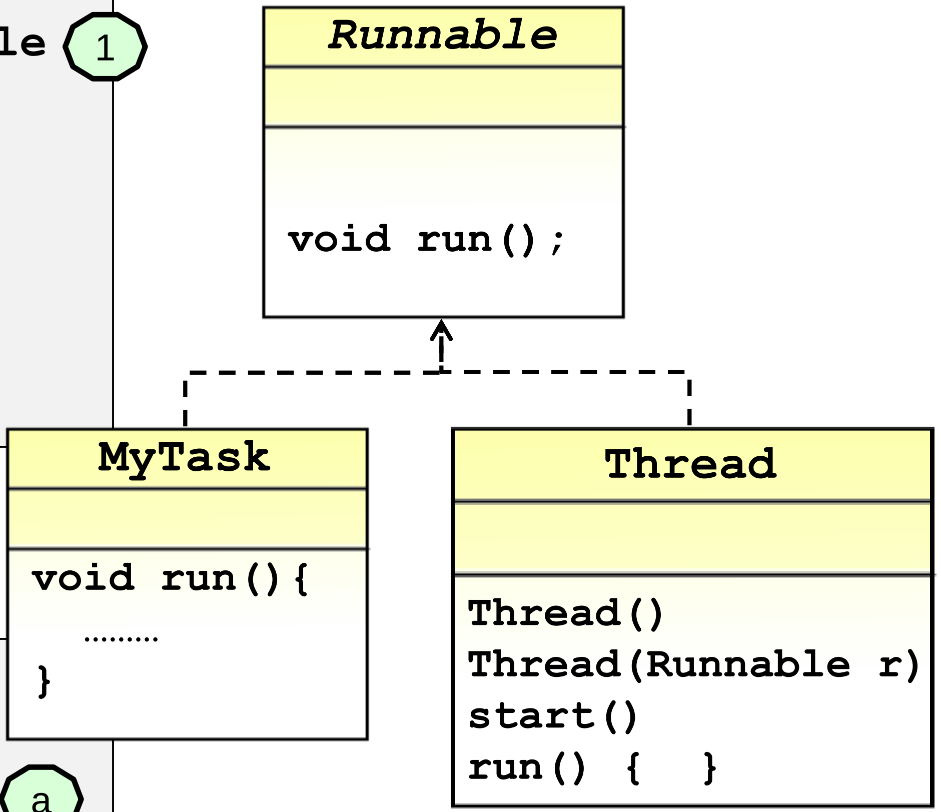
    1. Define a class that implements **`Runnable`**.
    2. Override its **`run()`** method .
    3. In main or any other method:
       a. Create an object of your class.
       b. Create an object of class **`Thread`** by passing your object to the constructor that requires a parameter of type **`Runnable`**.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

Working with Threads cont'd

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

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

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



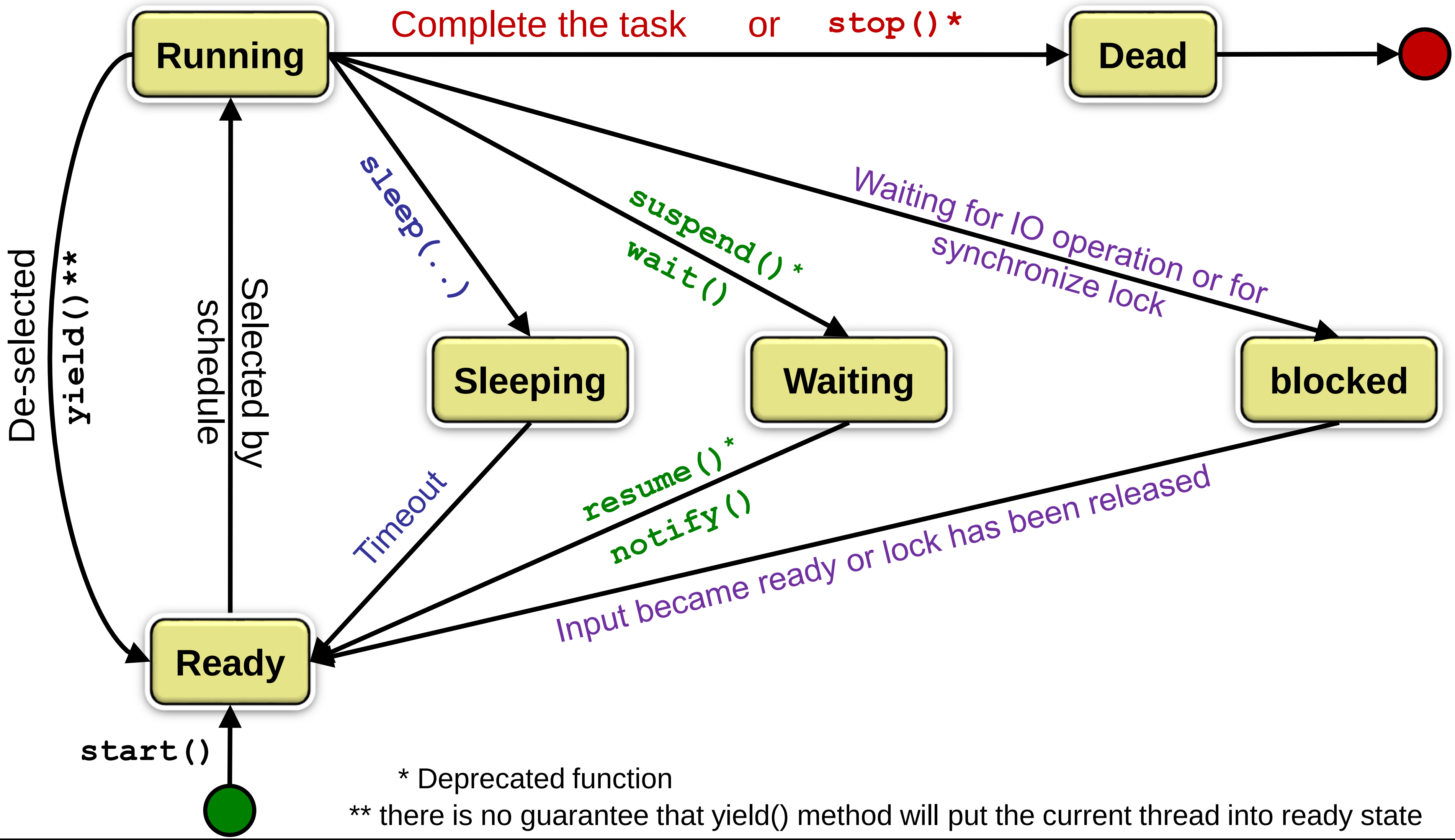
Extending Thread VS. Implementing Runnable

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

Example: DateTimeApplet

```
public class DateTimeApp extends Applet implements Runnable{
   Thread th;
   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



* Deprecated function

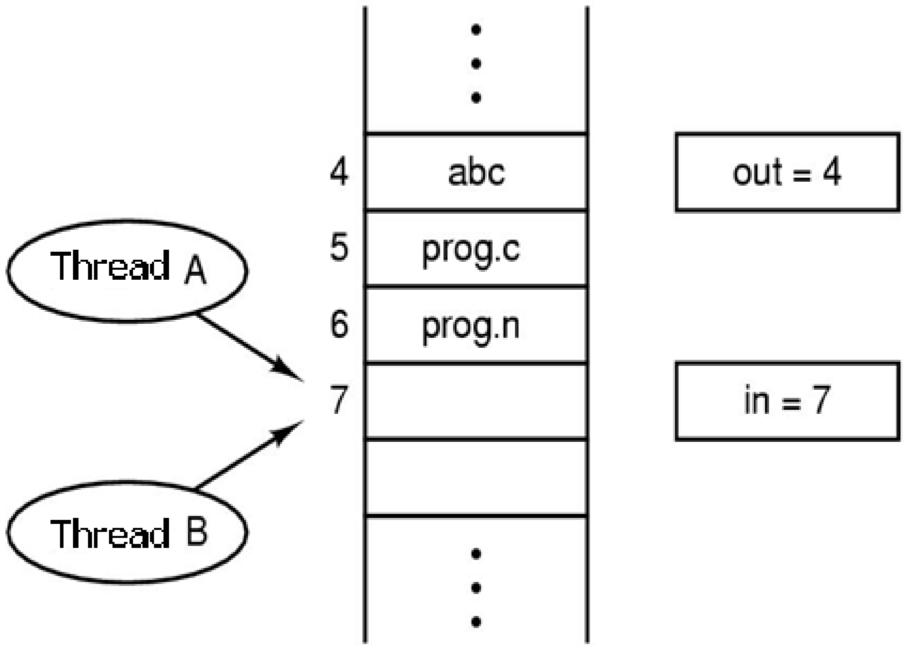
** there is no guarantee that yield() method will put the current thread into ready state

Synchronization

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

- Example:

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



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

Unsynchronized Example

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

- Sample output:

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

Synchronized method

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

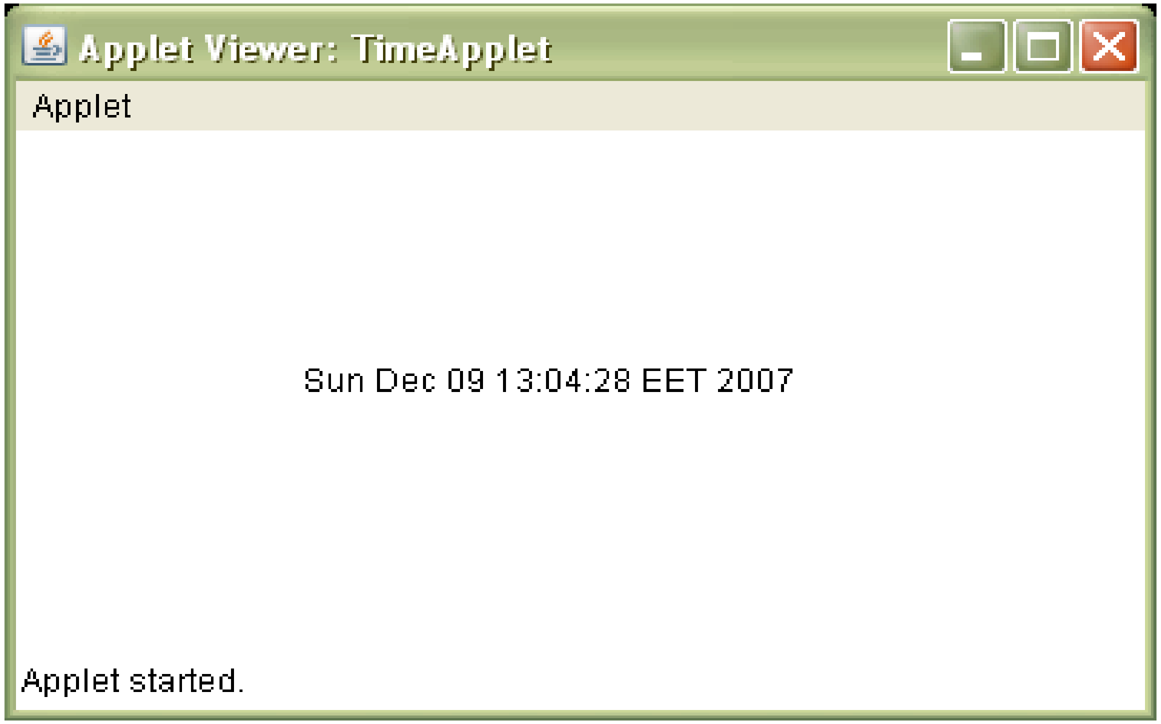
- Sample output:

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

Lab Exercise

1. Date and Time Applet

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

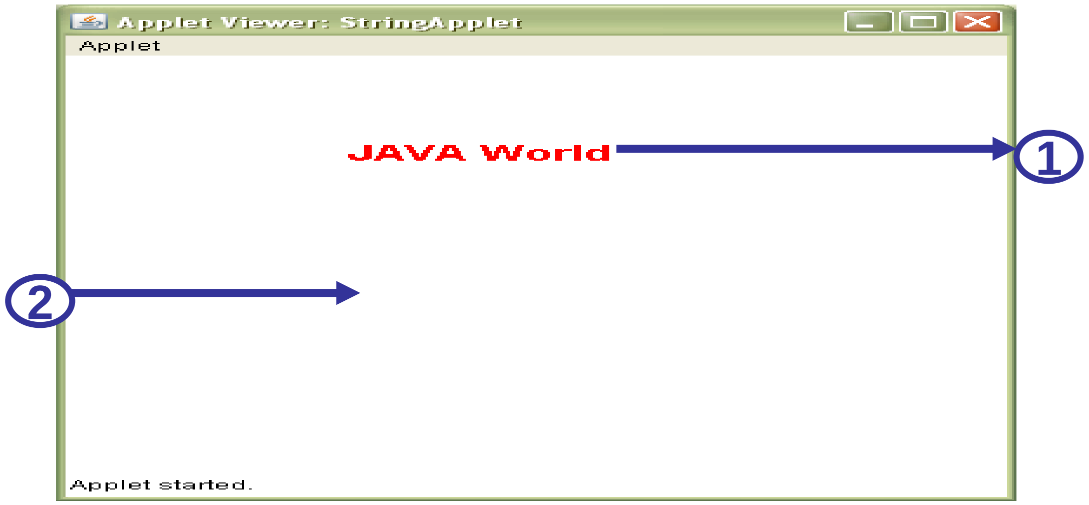


```
Sun Dec 09 13:04:28 EET 2007
```

Applet started.

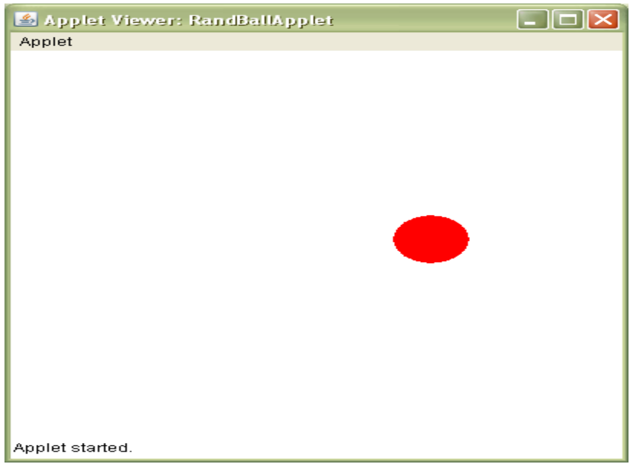
2. Text Banner Applet

•Create an applet that displays marquee string on it.



3. Animation Ball Applet

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


Applet Viewer: RandBallApplet
Applet
Applet started.