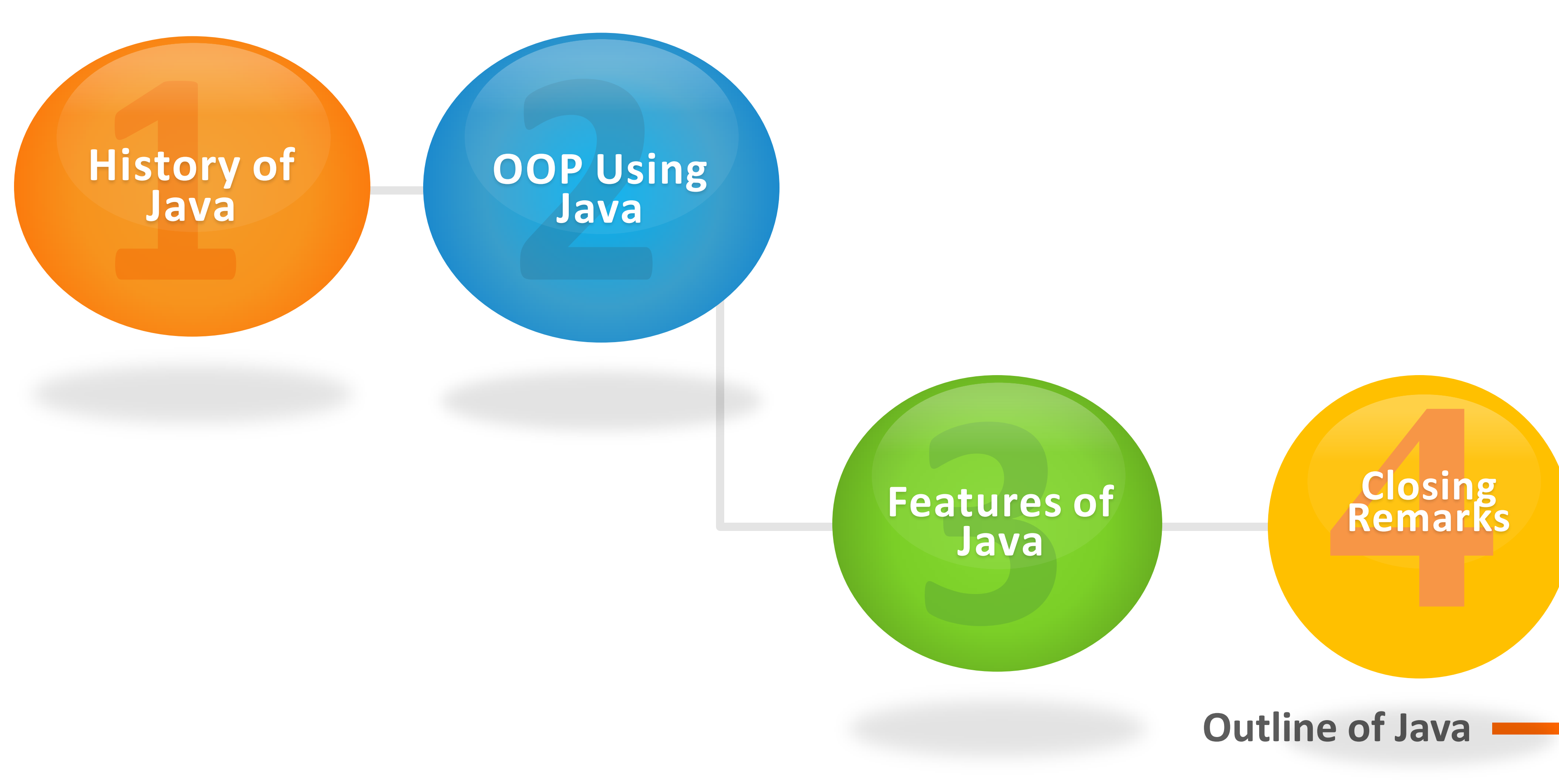
a guide to an object oriented programming language
Niket Sheth
JAVA

# Agenda




Outline of Java

# History of Java

Brief History Of How Java Came To Be

# History (I)

- The Java platform and language began as an internal project at Sun Microsystems in December 1990.
- Initiated due to the frustration caused by the complexity of C++ and its lack in services of security, portability, distributed programming, and multithreading.
- Patrick Naughton, James Gosling, and Mike Sheridan decided to create the new programming language instead of modifying and extending C++.

# History (II)

- Initially called *Oak,* the language and the platform failed to make an impression in the cable TV industry which forced the developers to re-launch product for the World Wide Web in 1994.

- A small browser was created by Naughton called WebRunner (later renamed HotJava) which was implemented using Java. The browser supported Java applets and served as a demonstration for Sun's new technology.

- BOOM! Java's potential was finally acknowledged in the World Wide Web and the rest, as they say, is history.

# OOP Using Java

OOP Using Java

# Overview

- Classes
- Objects
- Inheritance
- Interfaces

# Classes – Declarations

- Classes are a blueprint for an object
- Declared as follows:

```
1  public class ClassName
2  {
3      //class body
4      //i.e. attributes, constructors, methods
5  }
```

- Class declarations include:
  - Access modifies (discussed later) preceding the **class** keyword
  - The class name
  - The name of parent class (superclass, discussed later) preceded by the keyword **extends**.
  - A the name of the interface (discussed later) that the class represents preceded by the keyword **implements**. A comma-separated list if more than one interface.
  - The class body inside the braces {}

# Classes – Access Modifiers (I)

- To help increase the integrity of the software system built by developers, Java offers the use of modifiers to control the access to members defined in a class.

  - *public*: Members are accessible anywhere the class itself is accessible and are also inherited by all subclasses.
  - *protected*: Members accessible within its own class as well as its own package. Furthermore, accessible to and inheritable by code in subclasses in another package.
  - *private*: Members declared private are accessible only within their own class.

# Classes – Access Modifiers (II)

- When no modifier precedes a member declaration, a default modifier is supplied by Java of type *package-private*.
  - *package-private*: Member accessible and visible only within its own package.
- The table below summarizes the modifiers and their access levels:

| Modifier | Class | Package | Subclass | Outside World |
|---|---|---|---|---|
| *public* | ✓ | ✓ | ✓ | ✓ |
| *protected* | ✓ | ✓ | ✓ | X |
| *private* | ✓ | ✓ | X | X |
| *package-private* | ✓ | X | X | X |

# Classes – Constructors (I)

- Constructor is a body of code that is executed to initialize an object after its creation.
- Declared in the same fashion methods are declared, but they have the same name as the class they initialize with no return types.

```
1  public ClassName ([optional args])
2  {
3      //Initialization
4  }
```

- A class can have multiple constructors and each constructor can have zero or more arguments.

# Classes – Constructors (II)

```
ClassName myObject = new ClassName(args);
```

- The **new** operator in the sample above creates an object by invoking the constructor with the corresponding arguments to initialize the object.
- The **new** operator invokes the appropriate constructor based on the number of arguments that were provided to it.
- Java differentiates the multiple constructors based on the number and type of parameters in the list like it would with overloaded methods.

# Classes – Constructors (III)

- If no explicit constructor is defined in a class, Java provides a default constructor which takes no arguments and performs no special actions or initializations.

- This default no-argument constructor is of the superclass.

- Access modifiers can be used to control the call of the constructor in relation with other classes

# Classes – Destructors (I)

- Initializing objects as they are created seem easy, but how about destroying the objects?
- Some languages require the programmer to delete objects that are no longer in use to free up memory in the system.
- Java realizes the disadvantages of that concept and provides a utility that automatically takes care of destroying the objects.
- This internal destructor of Java is referred to as ***Garbage Collection*** and is a form of automatic memory management.
- The programmer has no control over this feature of when the object gets destroyed.

# Classes – Destructors (II)

- Garbage collector calls the **finalize()** method on an object when it determines that there are no more references to it.

- Destroying and freeing up memory is now the responsibility of the system, not the programmer.

- **GREAT!** No more memory leaks due to programmer's inability to delete objects when necessary or any accidents where an object is deleted even if it has references to it (referred to as the dangling pointer error).

# Classes – Nested Classes (I)

- Java allows the programmer to define a class within another class, called a **Nested Class**.
- Nested classes can be static or non-static. Non-static classes are called **Inner Classes**.
- Nested classes are defined the following way:

```
class OuterClass {
    //outer class body

    static class StaticNestedClass {
        //Static class body
    }

    class InnerClass {
        //Inner class body
    }

    //rest of outer class body
}
```

- **OuterClass** can access public or private members of the nested classes.

# Classes – Nested Classes (II)

- **Static Nested Classes**
  - Have class scope. Associated with its outer class.
  - Static nested classes are top-level classes. Definition is preceded by the **static** keyword followed by **class** and then then class name.
  - Can access ONLY static members of the outer class.
  - Use outer class members ONLY through an object reference.
  - Instance of outer class not needed to instantiate static nested class.
  - Static nested classes accessed via the following way:
    - **OuterClass.StaticNestedClass**
  - To create an object of the static nested class
    - **OuterClass.StaticNestedClass staticObj = new OuterClass.StaticNestedClass();**

# Classes – Nested Classes (III)

- **Inner Classes**
  - Instance scope (associated with an instance of its outer class).
  - CANNOT define static members since associated with an instance.
  - **InnerClass** instance exists ONLY WITHIN an **OuterClass** instance.
  - **InnerClass** instance has direct access to all the methods and attributes of **OuterClass**, even if they are defined to be private.
  - To instantiate an inner class, the outer class needs to be instantiated first and then create the inner class object within the outer class object

```
OuterClass outObj = new OuterClass();
OuterClass.InnerClass innerObj = outerObject.new InnerClass();
```

# Classes – Nested Classes (IV)

- **Why Nested Classes?**
  - *Logical Grouping of Classes* – If a class needs the help of another class, it is logical to embed it in the class and keep them grouped. This nest of helper classes simplifies the code.

  - *Increased Encapsulation* – If class B needs class A, then B can be defined in A.  That way while keeping A's members private, B can still access them and be hidden from the rest of the world.

  - *Increased Readability/Maintainability of Code* – Small nested classes within top-level outer classes place the code closer to where it is being used.

# Objects - Creation

- Objects are an instance of a class.
- Interact with other objects by invoking methods.
- Have their own copy of class attributes.
- Three parts to creating an object
  1. Declaration
     - **type** *objName*;
  2. Instantiation
     - The **new** operator creates the object by allocating memory for the object and its' attributes.
  3. Initialization
     - The **new** operator calls the constructor to initialize the object.  Constructor can have zero or more arguments.

1. 2. 3.

**ClassName objName = new ClassName(args);**

# Objects - Usage

- **Referencing Object's Attributes**
  - **objectReference.attributeName;**

- **Referencing Object's Methods**
  - **objectReference.methodName(args);**

    \- or -

  - **new ClassName(args).methodName(args);**

  - **ClassName** above refers to the constructor.

  - A method is invoked as long as it is acted upon by an object reference. The second way above returns an object reference and therefore the expression is valid.

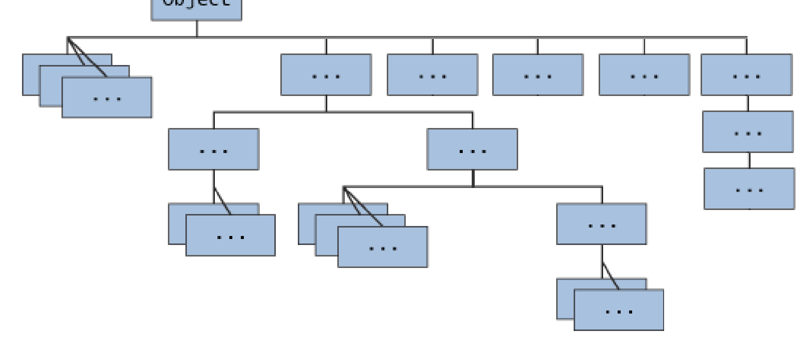
  - There can be zero or more arguments provided to the method.

# Objects - Deletion

- Some OO languages require the programmer to keep track of all the objects created and have them explicitly delete them.
- As mentioned before, Java provides a destructor feature (**Garbage Collection**) which keeps track of the object created and deletes it when it is no longer needed.
- This shifts the responsibility of deleting objects to the system and prevents any accidents or memory leaks from occurring.

# Inheritance – java.lang.Object

- **java.lang** package of the Java platform defines a class **Object** which implements behavior common to all classes and sits at the top of the class hierarchy tree. ***




- All classes in the Java platform (even the one the programmer creates) are subclasses of the **Object** class.
- The default no-argument constructor that Java provides for classes with no explicit constructor defined comes from the **Object** class since it has one defined in its class body.
- **java.lang.Object** provides methods that can be invoked on any Java object for utility and threading support (i.e. **finalize(), clone(), getClass()**).

*** The Java Tutorials

# Inheritance – Declaration and Overriding

- Inheritance is declared via the **extends** keyword which is placed after the class name and before the superclass name. It is declared the following way:

```
1  public SubClassName extends SuperClass {
2      // class body
3  }
```

- A subclass can define a method with the same method signature as the one in its superclass which will hide the method implementation in the superclass. This is called method **overriding**.
- Inheritance is a powerful OO technique, but can be dangerous if a subclass does not uphold the contract set forth by its superclass.
- One of the ways to avoid such a concern can be through Java's **final** modifier. When this modifier precedes a class or method declaration, it prevents the class from being subclassed and prevents the method from being overridden, respectively.

```
1  class CheckersGame {
2      enum CheckersPlayer {WHITE, RED}
3
4      final CheckersPlayer getFirstPlayer() {
5          return CheckersPlayer.RED;
6      }
7      //rest of class body
8  }
```

- Imagine a CheckersGame algorithm which goes through and evaluates the game. If there happened to be a subclass for this algorithm, we would not want the subclass to change the **getFirstPlayer()** method to return something different. Thus, it is declared with the **final** modifier to prevent it from it being overridden and exhibit unwanted results from such an action.

# Inheritance – **super** operator (I)

- Through inheritance, Java provides its developers with the use of the **super** keyword.
- **super** is for use in a subclass and it refers to the methods in the superclass.
- **super** is used to access all the non-private methods of a superclass.
- **super** knows nothing about the methods of a subclass when used. It ONLY knows about the methods in the superclass and executes them.
- We know that overriding methods in a subclass hides those same methods in the superclass instead of replacing them. So, to access those hidden methods, **super** can be used.
- **super** is handy when an overridden method in the subclass is not appropriate and actual behavior of the superclass method is necessary or a method in superclass is needed for use in the subclass definition.

# Inheritance – **super** operator (II)

- In this example, **Subclass** overrides **methodOne()**.
- Within the Subclass, **methodOne()** refers to the implementation defined in **SubClass** since it overrides the method declared in **SuperClass**. So to refer to the method defined **SuperClass**, the **SubClass** must use the proper name invocation using **super** as shown.
- Compiling and executing **Subclass** prints the following print statements in the command prompt.
- It can be seen that **super.methodOne()** invokes the **methodOne()** of **SuperClass** and executes its code.

```
1  public class Superclass {
2      //Superclass Method
3      public void methodOne() {
4          System.out.println("Method of Superclass");
5      }
6  }
7  public class Subclass extends Superclass {
8      //Subclass Method
9      public void methodOne() { //overrides methodOne in Superclass
10         super.methodOne(); //uses the methodOne of superclass which is overridden
11         System.out.println("Method of Subclass");
12     }
13     public static void main(String[] args) {
14         Subclass s = new Subclass();
15         s.methodOne();
16     }
17 }
```

Output when Subclass.java is compiled and executed. Shows how **super** is used to invoke superclass methods.

```
Method of Superclass
Method of Subclass
```

# Inheritance – **this** operator

- At times the object needs to know its own reference and so Java provides the use of **this** operator.
- It is generally used in the body of code of an instance method to refer to the object that contains the method.
- The intent is to refer to “this object”, the one right here that the method is in.

```
public class Employee {
    //Name of the employee
    private String name;
    //Constructor to create an employee with specified name
    public Employee(String name) {
        this.name = name;
    }
    //Rest of class body
}
```

- For example, in the constructor above, I need to assign a value to the instance field of the object that contains the constructor and so **this.name=name** does exactly that. It sets the **name** field of the object of the class by assigning the “value” of the **name** argument that was passed in to the constructor.

# Inheritance – Constructor Chaining

- A subclass inherits all of the public methods and attributes of its superclass except its constructors. However, a subclass can use its superclass' constructor by using the **super** operator followed by the arguments that need to be passed in if appropriate.

- Invocation of a superclass' constructor must be the first line in the subclass constructor and then the initialization code of that subclass follows.

- **this** can be used to access other constructors in the same class and must be the first line in the class's constructor.

- If a subclass constructor invokes a constructor of its superclass, either explicitly through **super** or implicitly by Java, one can see that there will be a whole chain of constructors called, all the way back to the constructor of **Object**. This is called **constructor chaining**.

# Inheritance – Constructor Chaining

- In the example to the right, **Subclass** inherits all the public fields of its superclass while adding its own fields. However, the constructor does not get inherited.
- With the use of **super**, the subclass can still access the superclass' constructor.
- **this** can be used to access other constructors in the same class for help in initializations.
- In **SubClass**, *Constructor 2* calls the **SuperClass** constructor while *Constructor 3* calls *Constructor 1* of **SubClass** for part of the initialization.

```java
public class Superclass {
    //Superclass Attributes
    public int attributeOne;
    public int attributeTwo;
    //Superclass Constructor
    public Superclass(int arg1, int arg2) {
        attributeOne = arg1;
        attributeTwo = arg2;
    }
}
public class Subclass extends Superclass {
    //Subclass Attributes
    public int subAttributeOne;
    public int subAttributeTwo;
    public float subAttributeThree;
    //Subclass Constructor 1
    public Subclass(int subArg1) {
        subAttributeOne = subArg1;
    }
    //Subclass Constructor 2
    public Subclass(int subArg2, int arg1, int arg2) {
        //call superclass constructor to initialize
        super(arg1, arg2);
        subAttributeTwo = subArg2;
    }
    //Subclass Constructor 3
    public Subclass(int subArg1, float subArg3) {
        //call first constructor of Subclass to initialize
        this(subArg1);
        subAttributeThree = subArg3;
    }
}
```

# Inheritance – Multiple Inheritance

- Java DOES NOT support multiple inheritance.
- Java creators wanted a simplified language than C++ that developers can grasp quickly. So they made a language similar to C++ without carrying over its complexities, hence to make it simple and easy to use.
- Since multiple inheritance causes confusion and many problems than it actually solves, it was taken out of the equation by the creators.
- The concept of multiple inheritance can still be achieved through the use of interfaces which will be discussed in later topics.

# Inheritance – Abstract Classes (I)

- Programmers, in the creation of their software system, might want to make the "design by contract" principle explicit. This is where abstract classes and interfaces are useful. Java supports both these OO concepts of Abstract Classes and Interfaces (which will be discussed in a later topic).
- Abstract classes are meant to be subclassed. Subclasses must provide the abstract method implementations. Abstract classes can also provide implementations to method behaviors which are common to all subclasses.
- Abstract classes are defined with the use of **abstract** preceding the **class** keyword
- Abstract methods are declared with the keyword **abstract** preceding the return type of the method

```
1  public abstract class ClassName {
2      //declaration of attributes, non-abstract methods
3      //and their implementation
4
5      //abstract method declaration
6      abstract void methodName( [optional args] );
7  }
```

# Inheritance – Abstract Classes (II)

- There are going to be times when a programmer defines a superclass which outlines the structure of a given abstraction (a generalized form) which in some respects is shared by all its subclasses, but leaves the subclasses to fill in the rest of the details.
- An abstract class would come into play for this type of a scenario.
- For example, say a user wants to create an application which draws different shapes (i.e. circles, square) and displays their data (i.e. area). All shapes would have some common methods and attributes, but the way they are drawn and the calculation of their data is specific to the different shapes.

```
1  public abstract class Shape {
2      public String color;
3      public Shape() { }
4      public move(int x, int y) {
5          //non-abstract method implementation
6      }
7      abstract public double draw();
8      abstract public void area();
9  }
10
11 public class Circle extends Shape {
12     void draw() {
13         //method for drawing circle
14     }
15     void area() {
16         //method for calculating area of circle
17     }
18 }
19 public class Square extends Shape {
20     void draw() {
21         //methods for drawing square
22     }
23     void area() {
24         //method for calculating area of square
25     }
26 }
```

# Interfaces - Declaration

- Another way to "design by contract" explicitly is through the use of interfaces which Java provides its users. Interfaces are important and useful when programmers want to define roles in a software system that can be played by its classes.

- Interface declaration composes of an access modifier, the keyword **interface**, the name of the interface, and the body of the interface. If an interface extends from several other parent interfaces, then the keyword **extends** with a comma-separated list of parent interfaces follows after the name of the interface.

```
public interface InterfaceName extends InterfaceOne, InterfaceTwo{
    //constant declarations
    double gravity = 9.80665;
    //method signature declarations
    void methodOne (int x, int y);
    int methodTwo (String s);
    //rest of the interface body
    }
```

- The interface body consists of only method signatures (not their implementations) followed by a semicolon. Method implementations are not provided. All methods in an interface are implicitly **public** so the explicit use of the modifier is not required. Interfaces can define constant values which are **public**, **static**, and **final**.

- A class implements an interface by having the **implements** clause in the class declaration. The class can implement more than one interface, so a comma-separated list follows the **implements** clause. The **implements** clause follows the **extends** clause (if any).

# Interfaces - Implementation

- As an example we will build a **Speak** interface for the subclasses of **countryPerson.**
- The subclasses implement the **Speak** interface by providing implementations to those methods declared in the interface. The method implementations are specific to the subclass.
- At times a programmer might want to expose only the functionality of an object to its user, its programming interface, without revealing its implementation. This exhibits the concept of encapsulation which is important in OOP. Interfaces are a great way to showcase a thorough description between what the application programming interface is and what isn't.
- This way the implementation of the objects can change without having the user be affected by it or have the user be dependent on it.

```
//Interface
public interface Speak {
    public void sayHello();
    void sayBye();
}
//Superclass
public class countryPerson {
    //Constructor w/ no arguments
    public countryPerson () {    }
    public String hairColor;
    public int height;
    public void sayFrom(){
        System.out.println("I am from a different country");
    }
}
//Subclass of countryPerson implements the interface
public class spainPerson extends countryPerson implements Speak {
    public void sayHello() {
        System.out.println("Hola");
    }
    public void sayBye() {
        System.out.println("Adios");
    }
}
public class indiaPerson extends countryPerson implements Speak {
    public void sayHello() {
        System.out.println("Namaste");
    }
    public void sayBye() {
        System.out.println("Aavjo");
    }
}
```

# Interface – Multiple Inheritance

- Since Java does not support multiple inheritance, a programmer can still reap the benefits of such concept through multiple interfaces in which a class can implement more than one interface.

- Since Java classes can implement any number of interfaces, the classes can play multiple roles. Due to this, a class can take advantage of the multiple inheritance scheme as they display behavior of the various interfaces it implements without having to deal with the complexities and difficulties of multiple inheritance.

- Implementation of more than one interface is done through a comma-separated list.

```
public class Shape implements Rotatable, Translatable, Scalable {
    //class body
}
```

- **Shape** would have to provide implementations for all the methods declared in all the interfaces it implements.

- As one can see, a developer can still take advantage of multiple inheritance by having **Shape** play several different roles via the interfaces it implements.

# Interfaces – Interface Inheritance

- An interface can extend any number of other interfaces through inheritance. This is referred to as **Interface Inheritance**.
- If an interface extends from several other parent interfaces, then the keyword **extends** with a comma-separated list of parent interfaces follows after the name of the interface.
- A class who implements Baseball must implement the methods defined by that interface as well as the methods inherited from its parent interfaces.

```
public interface Sports {
    void setHomeTeam(String name);
    void setAwayTeam(String name);
}
public interface WorldSeries {
    void cleanField();
    int ticketsSold();
    void foodVendors();
    void securityCheck();
}

public interface Baseball extends Sports, WorldSeries {
    void homeTeamScored();
    void awayTeamScored();
    void endOfInnings();
    void extraInnings();
}
```

# Features of Java

The Things That Make Java A Powerful OO Language

Features of Java

# Overview

- Annotations
- Anonymous Classes
- Generics
- Packages
- Enum Types
- Application Programming Interface (API)

# Annotations – What they are and for what?

- Java introduces many new features for its users, one of them being Annotations.
- Annotations offer a way to associate and add metadata with program elements like classes, interfaces, and methods.
- They serve as a systematic way to support a declarative programming model.
- Annotations provide data about a program without affecting the operation of the code they annotate.
- Some uses for the Java feature of annotations:
  - *Information for the Compiler* – Annotations used by the compiler to detect errors or suppress warnings.
  - *Compiler-Time and Deployment-Time Processing* – Tools can process annotations to generate code, XML files, and etc.
  - *Runtime Processing* – Annotations can also be examined at runtime.
- The inclusion of annotations leads developers to create programs that are less likely to be bug-prone.

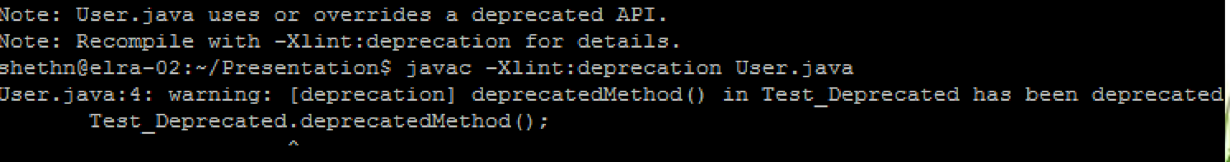
# Annotations – Three Predefined by Java (I)

## @Deprecated

- As a software system evolves due to new requirements, changes to its API are expected (i.e. new and better methods added, methods renamed, attribute fields change). Such changes introduce problems because the old API needs to be kept around until developers make the transition to the new API and not continue to program the old one.
- So to illustrate this deprecation of the old API, **@Deprecated** can be used. A developer can add an **@deprecated** (notice the lower case 'd') tag to a method's javadoc comments for an explanation of why it was deprecated.
- The program element marked with this annotation informs the compiler that the element is deprecated and should not be used. The compiler then produces a warning when an element is used with the **@Deprecated** annotation.
- **@Deprecated** is put above the deprecated fields or methods as shown. The warning as a result of compiling a file that uses **@Deprecated** is shown.

```
public class Test_Deprecated {
    // Javadoc comment follows
    /**
    * @deprecated
    * explanation of why it was deprecated
    */
    @Deprecated
    public static void deprecatedMethod() {
        System.out.println("Don't call me");
        //rest of method body
    }
}
```

```
public class User {
    public static void main(String args[]) {
        System.out.println("HELLO!");
        Test_Deprecated.deprecatedMethod();
    }
}
```



# Annotations – Three Predefined by Java (II)

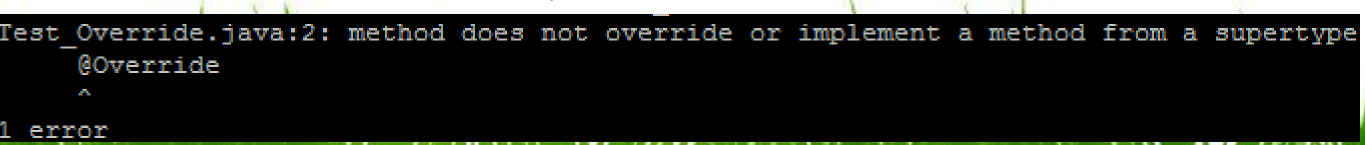
## @Override

- This annotation is meant to be used with method declarations in a program. This annotation informs the compiler that the method is required to override a method of the superclass. Compiling a program with this annotation would produce a compile-time error notifying the user if the method did not override its superclass method.
- This annotation helps capture errors quickly. Programmers can make mistakes and misspell the method name, specify the wrong arguments, or have a different return type when trying to override an existing method. This helps considerably with the debugging phase if such an issue were to occur since the compiler would generate the error to let the developer know.
- **@Override** is put above the method that needs to be overridden as shown:

```
1  @Override
2  public void overridenMethod() {
3      //method body
4  }
```

- Example: The example shows how the user misspelled the hashCode() function of Object class and received an error.

```
public class Test_Override {
    @Override
    public int hashcode() {
      return 5448;
    }
}
```



# Annotations – Three Predefined by Java (III)

## @SuppressWarnings

- This annotation informs the compiler to suppress specific warnings that it would normally tell you about.
- In Java, warnings belong to a certain category, so one has to tell the annotation which type of warnings to suppress.
- For example, if you want to use a deprecated method, the compiler would normally produce a warning. However, **@SuppressWarnings** annotation causes such a warning to be suppressed.
- The category of which warnings you want to suppress are put in parenthesis followed by braces as shown. More than one warning type is done through a comma-separated list in those braces. The annotation is put above the expression you don't want the compiler to generate a warning about.

**@SuppressWarnings( {warning1, warning2} )**

```
1  //Annotation used to tell the compiler
2  //not to generate a warning for using
3  //a deprecated method.
4  @SuppressWarnings({"deprecation"})
5  public void useDeprecatedMethod() {
6      obj.DeprecatedMethod();
7      //rest of the method body
8  }
```

- The example from **@Deprecated** is used to show how the compiler does not generate a warning when **@SuppressWarning** is used with **deprecation** as the type of warning.

```
public class Test_Deprecated {
    // Javadoc comment follows
    /**
    * @deprecated
    * explanation of why it was deprecated
    */
    @Deprecated
    public static void deprecatedMethod() {
        System.out.println("Don't call me");
        //rest of method body
    }
}
```

```
public class User {
    @SuppressWarnings({"deprecation"})
    public static void main(String args[]) {
        System.out.println("HELLO!");
        Test_Deprecated.deprecatedMethod();
    }
}
```

```
HELLO!
Don't call me
```

# Anonymous Classes – What, When, Why?

**What?**

- Anonymous Classes are classes without names (hence anonymous).
- They are defined and instantiated in a succinct manner using the **new** operator.
- **Anonymous Class** is a Java expression where defining the class and instantiating it are combined into one step. Since this type of a definition is an expression it can be included as part of a larger Java expression, like a method call.
- Methods defined in anonymous classes have direct access to all of the members of the enclosing class, even private members.

**When?**

- Anonymous classes should be used when
  - The class body is short
  - Only one instance of the class needs to be instantiated
  - The class is going to be used right after it's declaration

**Why?**

- Improves readability of code since syntax is succinct and reduces clutter in code.
- Lets developers define one-shot, "on the fly" classes where they are needed when a specific event takes place.
- Is a good way to group classes together

# Anonymous Classes –Declaration & Rules

## Declaration

- The syntax of how to define and instantiate an anonymous class is shown in the figure below:

```
1  new ClassName ( /*optional args*/ ) { /*class body*/ }
2
3          -- or --
4
5  new InterfaceName () { /* interface body */ }
```

- Either of the two expressions can be used and they are usually incorporated in a larger overall Java expression (like a method call).

## Rules

- Anonymous classes cannot be public, private, protected, or static.  No modifiers specified in anonymous class definitions.
- Only one instance of an anonymous class can be created since the definition is a single "in-line" expression.  Anonymous classes not appropriate for creation of more than one instance of the class whenever the code is executed.
- Cannot define constructors in anonymous classes.

# Anonymous Classes – Example

- Anonymous classes are most often used in handling events in a GUI. So rather than defining a new class and then instantiating an object from it to handle each event of related to a GUI (press of a button in a menu), anonymous classes would be used.
- In the example below, to do processing on a button in the GUI, we implement ActionListener for the button we want to listen to and define the class and instantiate the object right on the spot while defining and calling a method for it.
- Notice the fact that there is no name for the class when it is defined. It starts with the **new** keyword. **ActionListener()** is the interface name and implementation for its method is provided.
- The semicolon at the end of the statement is actually part of the class definition and required at the end of every Java expression.

```
public class exampleGUI extends JFrame {
    //button member declarations
    protected void makeGUI() {
        button = new JButton();

        button.addActionListener (new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("You pressed a button");
            }
        });
    }
}
```

# Generics – What, When, and Why?

## What?

- Generics are a feature that define generic types and methods. The generic types or methods are composed of type parameters.
- Allow developers to write generic code independent of a particular type.  This is similar to the template feature in C++.
- Generic types form parameterized types by providing actual type arguments to replace the formal type parameters.

| | |
|---|---|
| *ArrayList<E>* | Generic type with type parameter *E* |
| *ArrayList<String>* | Parameterized type with actual type argument *String* |

## When?

- Mostly when working with the implementation and use of collections (i.e. ArrayList, LinkedList)

## Why?

- Enables an early error detection of bugs at compile time instead of at run-time.
- Requires fewer type casts and allows the compiler to carry out more type checks.

# Generics – Generic Type

- If you define a generic class as defined below, it basically makes T a placeholder for a type that the user wants to later concretely define. T is a variable whose value is anything that the user passes in (i.e. class type, interface type). T is a formal type parameter.
- In the class declaration, it defines a private attribute of type T and the getter and setter methods return a value of type T.

```
public class ToolKit<T> {
    private T t;

    public void set(T t) {
        this.t = t;
    }
    public T get(){
        return T;
    }
}
```

- In order to create a reference to this generic class, the user has to do a **generic type invocation** where T is overridden to be an actual type, like String.
- To instantiate this class
  - **ToolKit<String> strToolKit = new ToolKit>();**
  - **strToolKit** is now referencing a "ToolKit of String"
- Furthermore, if the user invokes **set** with an incompatible type, like **Integer**, a compile error would be generated.

# Generics – Parameterized

- Let's say you are using a simple linked list stored with **Integers** of data. You want to retrieve all the elements from it and do more processing. The linked list is defined as follows.

```
LinkedList list = new LinkedList();
list.add(new Integer (1));
list.add(new Integer (2));
list.add("abc");
```

- Now if you want retrieve a specific part of the list, you'll need to do type casting since collections accept objects of various types but return an **Object** reference. Now if the user wants the third Integer, **Integer i=(Integer) list.get(2);** would be overlooked during compile time but at run-time **CLassCastException** will be thrown since we have stored a **String** at that location.
- If the list is designed with generics in mind, the bug would have been caught during compile time instead of run-time.
- Use of generics suppresses the use of casts since objects with a different reference other than the parameterized type (shown below between **< >**) used in generics would produce a compile time error.

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer (1));
list.add("abc");//generates error at compile time
String s = list.get(0);  //cast not needed
```

# Packages – The What

## What?

- As software developers, we tend to start with small projects where we put all our files under one directory. However, as our project grows bigger and bigger, putting all the files under one directory would not be a great idea and managing them would be a nightmare.
- To avoid such a crisis, Java gives us the feature of using **Packages**.
- Packages help organize a developer's java files into directories under the right categories according to their functionality and usability. It is a mechanism for grouping related types (i.e. classes, interfaces) into one common bundle.
- The basic concept is that files in one package have a different functionality than those in another package.
- For example, **java.io** package deals with I/O related services of reading and writing classes while **java.net** deals with providing users with classes that implement networking applications.
- As you can see, various packages of the Java platform are grouped by the function of their classes.

# Packages – The Why

## Why?

- Packages provide the ease of maintenance, organization, and improved collaboration amid developers.

- They are great for avoiding naming conflicts since packages create a new namespace.
  - For instance, if you have a class named **Calendar()**, this name would be in conflict with the **Calendar()** class defined in the JDK. However, **java.util** is the package for the **Calendar()** class in JDK. So putting your **Calendar()** class in another package you define prevents the name collision from occurring.

- Packages provide control access over your defined types.
  - Types within a package can have unrestricted access to other types in the same package, but a developer can still restrict access of those types outside the package.

- Packages offer the simplicity of finding and using the specific types of a software system.
  - Since packages are a mechanism for grouping related types (i.e. classes, interfaces) into a common bundle, it is easier for developers to know which types are related and where to find specific ones with a certain functionality.

# Packages – Creation

- Packages are defined with the keyword **package** followed by the name for the package at the top of the source file before any type definitions. It's that simple!
- Packages are for related types. In the Abstract Class section earlier we defined a bunch of classes that represented graphical shapes. Let's say we write an interface **Transformable** which transforms the shapes in a particular way which the classes can implement. The interface and classes are in their respective .java files. Since they are all related, let's bundle them up in a package. The way the package statement is written is shown at the right.

```
1   //in Transformable.java
2   package shapes;
3   public interface Tranformable {
4       //body
5   }
6
7   //in Shape.java
8   package shapes;
9   public abstract class Shape {
10      //body
11  }
12
13  //in Circle.java
14  package shapes;
15  public class Circle extends Shape implements Transformable {
16      //body
17  }
18
19  //in Sqaure.java
20  package shapes;
21  public class Square extends Shape implements Transformable {
22      //body
23
24  //in Triangle.java
25  package shapes;
26  public class Triangle extends Shape implements Transformable {
27      //body
28  }
```

# Packages – Usage (I)

- The types composed in a package are what's known as **package members.**
- To use a package member for writing code within the same package then those members can be referred to by their simple name such as **Circle** or **Square**.
- To use a member from a different package, there are three ways to refer to and use the member

1) Refer to the member by its fully qualified name
   - If the package has not been imported, you must use the fully qualified name of the member you want access that member. The following shows you the full qualified name of for the **Circle** class in the **shapes** package.
     - **shapes.Circle**
   - To create an instance of that member
     - **shapes.Circle circleObj = new shapes.Circle();**
2) Import the package member
   - To use a member you can just import the specific package member into your file.
   - Use the **import** keyboard followed by the package name, dot, and the specific member you want to use. Import statement is at the beginning of the file.
     - **import shapes.Circle;**
   - After the import statement, you can refer to **Circle** by its simple name
     - **Circle circleObj = new Circle();**

# Packages – Usage (II)

3) Import the member's entire package

- Instead of importing a specific member at a time, a user can import the entire package and all its types.
- To import the entire package just use the **import** keyword along with the package name, dot (.), and an asterisk (*).
  - **import shapes.*;** //import all classes from the shapes package
- With this statement, the user can refer to any class or interface in the **shapes** package by their simple name.
  - **Circle circleObj = new Circle();**
  - **Square squareObj = new Square();**

# Enum Types – The What

- Enum Types is a type whose fields compose of fixed constants. Enum types are implicitly **final**. Nested enum types are implicitly **static**.
- Since they are constants, the fields of the enum type are in ALL CAPS.
- They are used whenever a programmer is in need to denote a fixed set of constants like data points, choices on a menu, days of the week, and etc.
- You can define an enum type with an access modifier, then the **enum** keyword, followed by the name you want to refer to for your definition of the fixed set of constants. An example is shown below.

```
public enum Compass {
    NORTH, SOUTH, EAST, WEST
}
```

- Java's enum types are much more effective than the ones used by other languages. The **enum** keyword defines a class which can include not only constants, but methods and other attributes as well.
- Enums are extended from the **java.lang.Enum**. So enum cannot extend any other class, but still is able to implement interfaces. **Java.lang.Enum** offers its programmers special methods to invoke on the enum types such as **name()** which returns the of the enum constant and **equals(Object o)** which returns true if the specified object is equal to the enum constant.
- The compiler automatically generates some special methods when enums are created. One such method is the **values()** method which returns an array of all the constants of the enum type in the order they are defined. This special method can be used with a for loop to iterate over all the constants of the enum to perform some type of processing.
- Use of Enums provides a cleaner implementation for a group of constants.
- Enums can be defined separately or nested within a class.

# Enum Types – The How

```
public enum Football{
    //TEAM                      (Wins, Loses, PA)
    NEW_ENGLAND_PATRIOTS        (14, 2, 313),
    ATLANTA_FLACONS             (13, 3, 288),
    BALTIMORE_RAVENS            (12, 4, 270),
    PITTSBURG_STEELERS          (12, 4, 232),
    CHICAGO_BEARS               (11, 5, 286),
    NEW_ORLEANS_SAINTS          (11, 5, 307),
    NEW_YORK_JETS               (11, 5, 304),
    GREEN_BAY_PACKERS           (10, 6, 240);

    private final int wins;
    private final int loses;
    private final int pointsAgainst;

    Football(int w, int l, int ptsAgainst) {
        this.wins = w;
        this.loses = l;
        this.pointsAgainst = ptsAgainst;
    }

    public int wins()    { return wins; }
    public int loses()   { return loses; }
    public int pointsAgainst() { return pointsAgainst; }

    public double ptsPerGame() {
            return ( pointsAgainst / (wins + loses) );
    }

    public static void main(String[] args) {
        for (Football f : Football.values()){
            System.out.printf("%s had PTS/G of %f%n",
                                        f, f.ptsPerGame());
        }
    }
}
```

- In this example, each enum is defined the team's wins, loses, and Points Against (PA) stats. They get passed in to the constructor when the constants get created. A rule for enum types is such that constants must be declared first and then field and methods are defined (if any). At the end of the constant declaration, there should be a semicolon.
- The Football enum defines its own fields and methods which calculate the points per game of each team. This program iterates through each constant using a for loop via the **values()** method and outputs the points per game stat of that team with the calculation method built in the enum declaration called **ptsPerGame()**.
- When **java Football** is executed from the command prompt, the following output is displayed.

```
shethn@elra-02:~/5448$ java Football
NEW_ENGLAND_PATRIOTS had PTS/G of 19.000000
ATLANTA_FLACONS had PTS/G of 18.000000
BALTIMORE_RAVENS had PTS/G of 16.000000
PITTSBURG_STEELERS had PTS/G of 14.000000
CHICAGO_BEARS had PTS/G of 17.000000
NEW_ORLEANS_SAINTS had PTS/G of 19.000000
NEW_YORK_JETS had PTS/G of 19.000000
GREEN_BAY_PACKERS had PTS/G of 15.000000
```

# The Java API

- The Java Application Programming Interface (API) provides Java users with features, some of which we covered like java.lang.Enum and java.lang.Object, to help them with their object oriented programming.

- The API is composed of everything, from collection classes to GUI classes.

- Since programmers are generally lazy, Java has produced an API that saves a programmer both time and effort in implementing their software design.

- Instead of writing an implementation to calculate the square root of a number or want to read in a file, using the API classes of Java can provide a simple and worrisome implementation and execution of both those things.

- As mentioned before, all the classes in the Java API are grouped by relation and broken down into packages in which the **import** keyword is used to access them. There are packages that are useful for creating applets, for data transfer, for rendering and modifying images, for security frameworks, as well as packages for thread support, to name a few.

- So as a developer, it is always important to keep a reference of the Java API handy for programming purposes.

# Closing Remarks

# Last Words…

- It can be seen how Java is a simple language for any person to grasp, whether they are an expert in OOP or are just trying it for the first time.

- Classes, Objects, Inheritance, and Interfaces are easy to define and implement.

- Java doesn’t leave its developers hanging and offers them a wide variety of predefined classes in its API to assist them in their programming tasks.  These features save the time and effort of a developer in the implementation of a software system.

# Resources

The following sites were used for reference while making this presentation…

- *The Java Tutorials*: *http://download.oracle.com/javase/tutorial/java/index.html*
- *Java Language Specification*: *http://java.sun.com/docs/books/jls/third_edition/html/classes.html*
- *Object-Oriented Programming in Java:* *http://docstore.mik.ua/orelly/java-ent/jnut/ch03_01.htm*
- *Object Oriented Programming*: *School of Computer Science, University KwaZulu-Natal (PDF)*
- *Java*: *http://en.wikipedia.org/wiki/Java_%28programming_language%29*
- *Intro to Java*: *http://www.cs.colorado.edu/~kena/classes/5448/s11/lectures/10_introtojava.pdf*
- *Introduction to Java and Object Oriented Programming for Web Applications*: http://www.devdaily.com/java/java_oo/
- *Introducing Annotations*: *http://www.java-tips.org/java-se-tips/java.lang/introducing-annotations.html*
- *The Essence of OOP using Java, Anonymous Classes:* *http://www.developer.com/java/other/article.php/3300881/The-Essence-of-OOP-using-Java-Anonymous-Classes.htm*

Niket Sheth