Saraswati Vandana

या कुन्देन्दु तुषार हार धवला
या शुभ्र वस्त्रान्विता ।
या वीणा वर दंड मंडितकरा
या श्वेत पद्मासना ॥

या ब्रह्मा अच्युत शंकर प्रभृतिभिः
देवै सदा पूजिता ।
सा मां पातु सरस्वती भगवती
निःश्येश जाड्यापह ॥

Faculty of Engineering & Technology
Sankalchand Patel College of Engineering, Visnagar

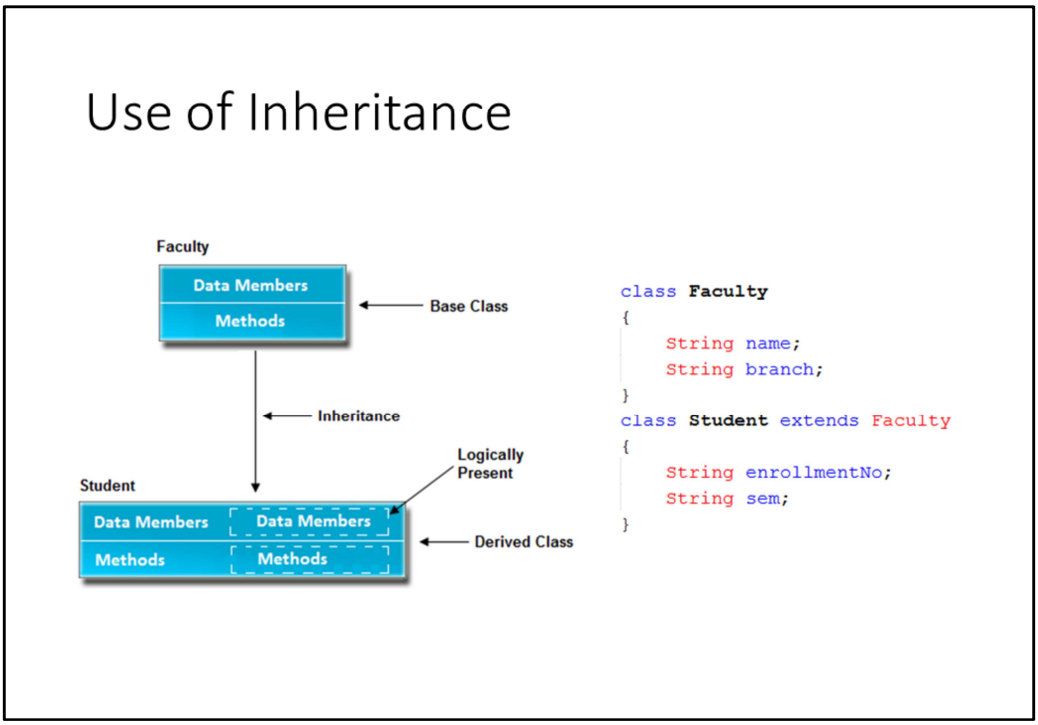
Java Programming
(1ET1030406)

Unit-4 : Inheritance, Interface & Polymorphism

Prepared By
Mr. Mehul S. Patel
Department of Computer Engineering & Information Technology

Content

- Use of Inheritance
- Inheriting Data members and Methods
- Constructor in inheritance
  - Super keyword
- Types of Inheritance
- Method overriding
- Stop Inheritance
  - Final keywords
- Abstract Class
- Creation and Implementation of an interface
- Multiple Inheritance using Interface
- Interface Inheritance
- Dynamic method dispatch
- Instance of operator
- Understanding of Java Object Class
- Comparison between Abstract Class and interface



Inheritance in Java

The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

Important points

- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.
- The data members and methods of a class are known as features.
- The concept of inheritance is also known as re-usability or extendable classes or sub classing or derivation.

Why use Inheritance ?

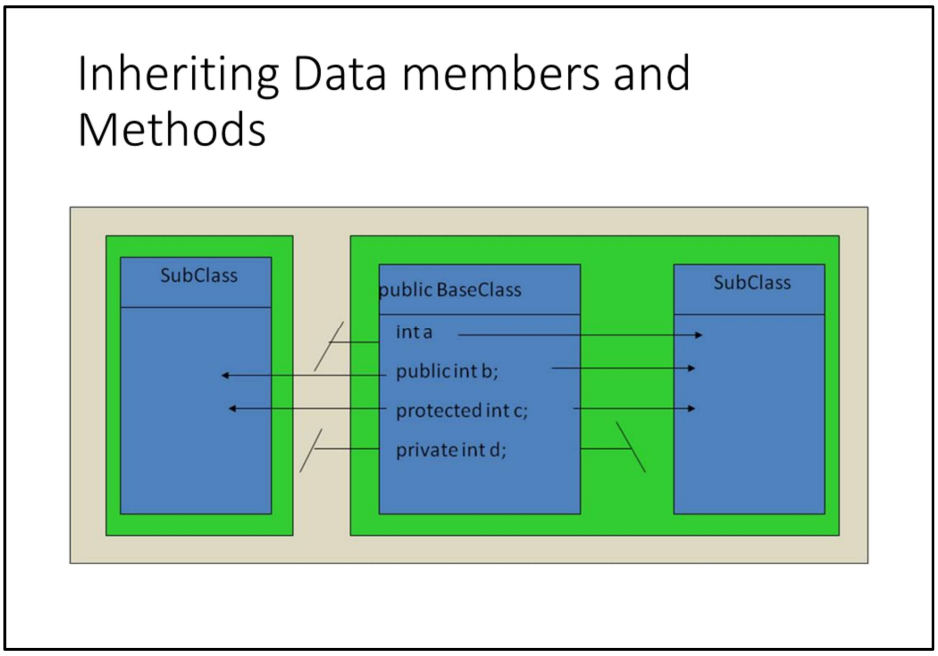
- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

Advantage of inheritance

- If we develop any application using concept of Inheritance than that application have following advantages,
- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.



Basic Access Modifiers

By default, the variables and methods of a class are accessible to members of the class itself and to other classes in the same package. To borrow from C++ terminology, classes in the same package are *friendly*. We'll call this the default level of visibility. As you'll see as we go on, the default visibility lies in the middle of the range of restrictiveness that can be specified.

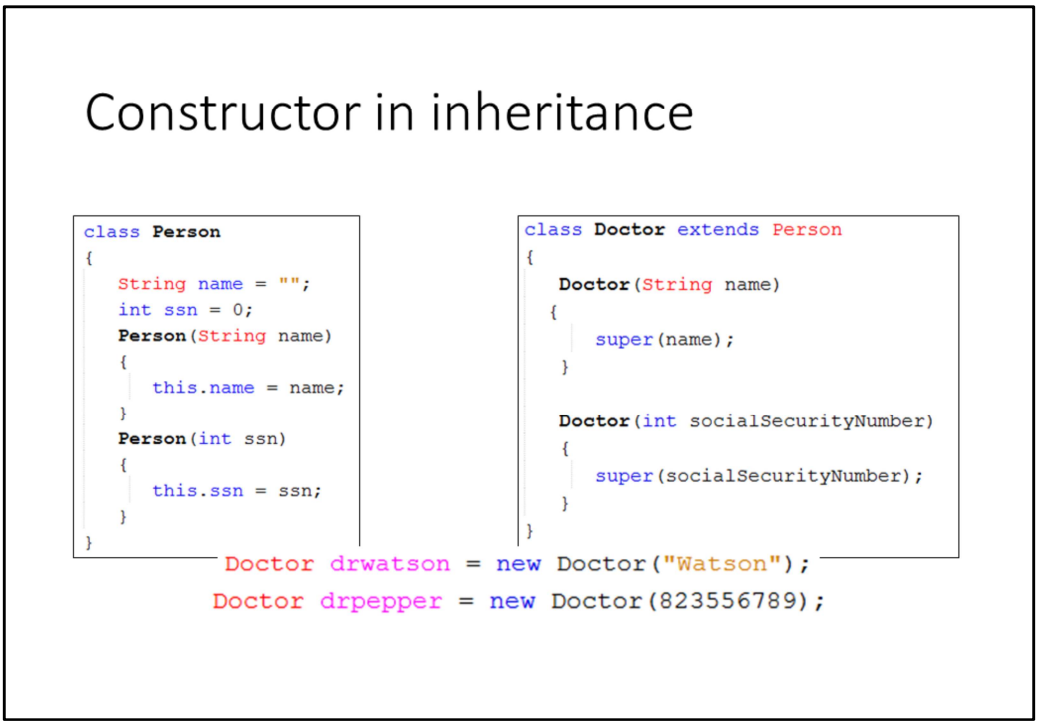
The modifiers public and private, on the other hand, define the extremes. Methods and variables declared as private are accessible only within their class. At the other end of the spectrum, members declared as public are accessible from any class in any package, provided the class itself can be seen. (The class that contains the methods must also be public to be seen outside of its package.) The public members of a class should define its most general functionality—what the black box is supposed to do.

Above figure illustrates the four simplest levels of visibility, continuing the example from the previous section. Public members in class are accessible from anywhere. Private members are not visible from outside the class. The default (not mentioned) visibility allows access by other classes in the package.

The protected modifier allows special access permissions for subclasses. Contrary to how it might sound, protected is slightly less restrictive than the default level of accessibility. In addition to the default access afforded classes in the same package, protected members are visible to subclasses of the class, even if they are defined in a different package.

summarizes the levels of visibility available in Java; it runs generally from most to least restrictive. Methods and variables are always visible within a declaring class itself, so the table doesn't address that scope.

Modifier	**Visibility outside the class**
private	None
No modifier (default)	Classes in the package
protected	Classes in package and subclasses inside or outside the package
public	All classes



Constructor in inheritance

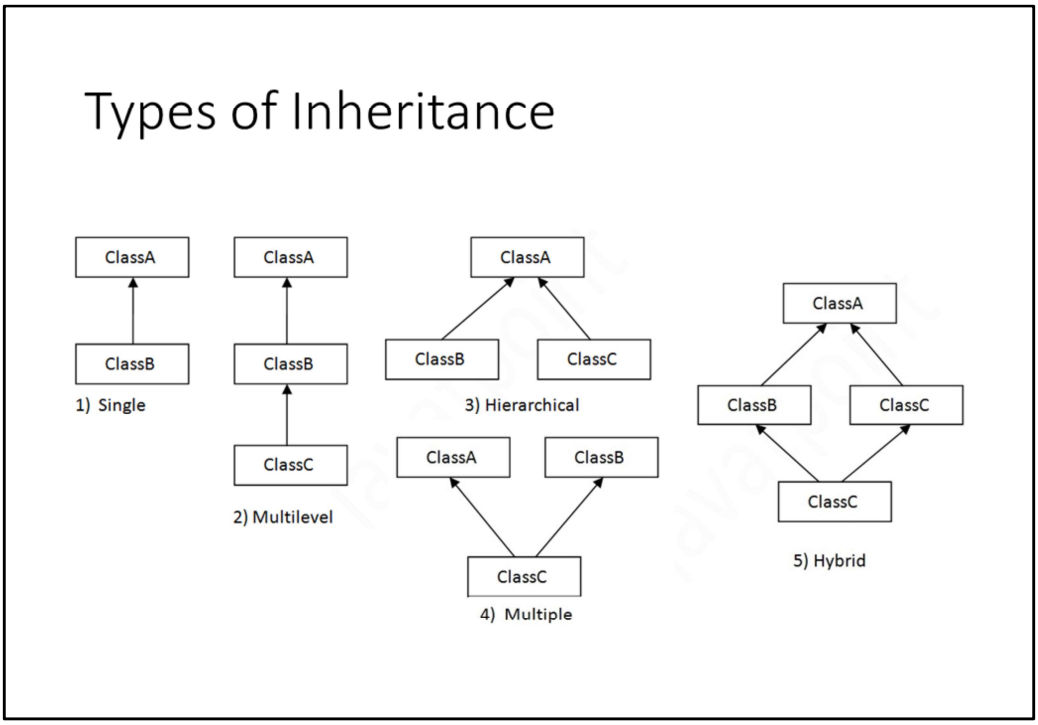
Subclasses inherit all the private instance variables in a superclass that they extend, but they cannot directly access them if they are private. And also constructors are not inherited.

How do you initialize inherited private variables if you don't have direct access to them in the subclass? In Java you can put a call to the parent constructor using as the **first line in a subclass constructor**. In Java, the superclass constructor can be called from the first line of a subclass constructor by using the keyword **super** and passing appropriate parameters, for example super(); or super (patameter);. The actual parameters given to super() are used to initialize the inherited instance variables, for example the name instance variable in the Person superclass.

If a class has no constructor in Java, the compiler will add a no-argument constructor. A no-argument constructor is one that doesn't have any parameters, for example public Person().

If a subclass has no call to a superclass constructor using super as the first line in a subclass constructor then the compiler will automatically add a super() call as the first line in a constructor. So, be sure to provide no-argument constructors in parent classes or be sure to use an explicit call to super() as the first line in the constructors of subclasses.

Regardless of whether the superclass constructor is called implicitly or explicitly, the process of calling superclass constructors continues until the Object constructor is called since every class inherits from the Object class.

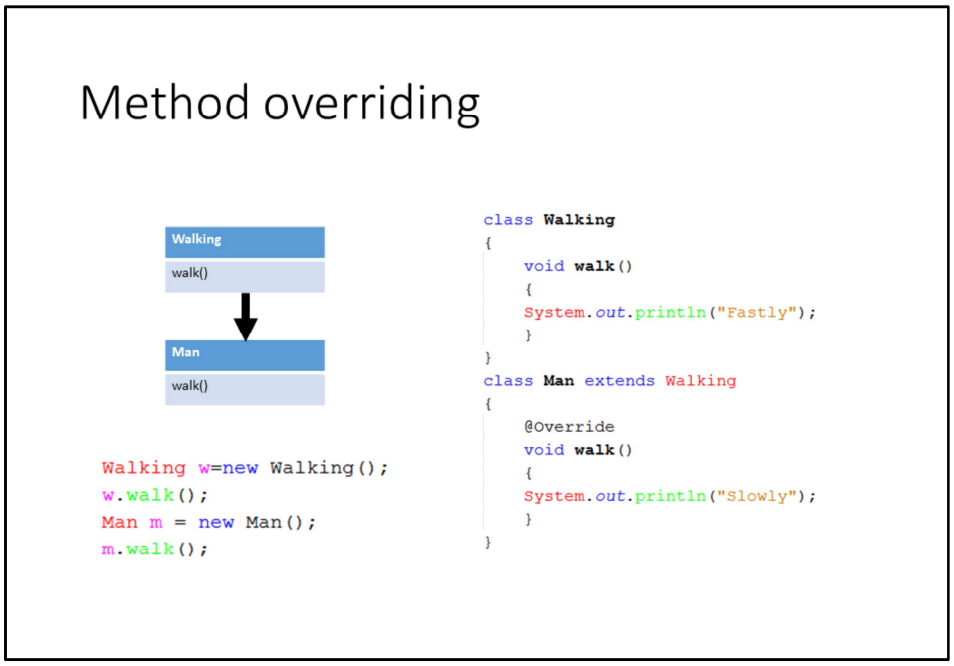


Tpyes of Inheritance

Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance; they are:

1) Single inheritance
2) Multilevel inheritance
3) Hierarchical inheritance
4) Multiple inheritance
5) Hybrid inheritance

1. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In the above image, class A serves as a base class for the derived class B.
2. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the above image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.
3. **Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the above image, class A serves as a base class for the derived class B, and C.
4. **Multiple Inheritance (Through Interfaces):** In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces. In the above image, Class C is derived from interface A and B.
5. **Hybrid Inheritance(Through Interfaces):** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



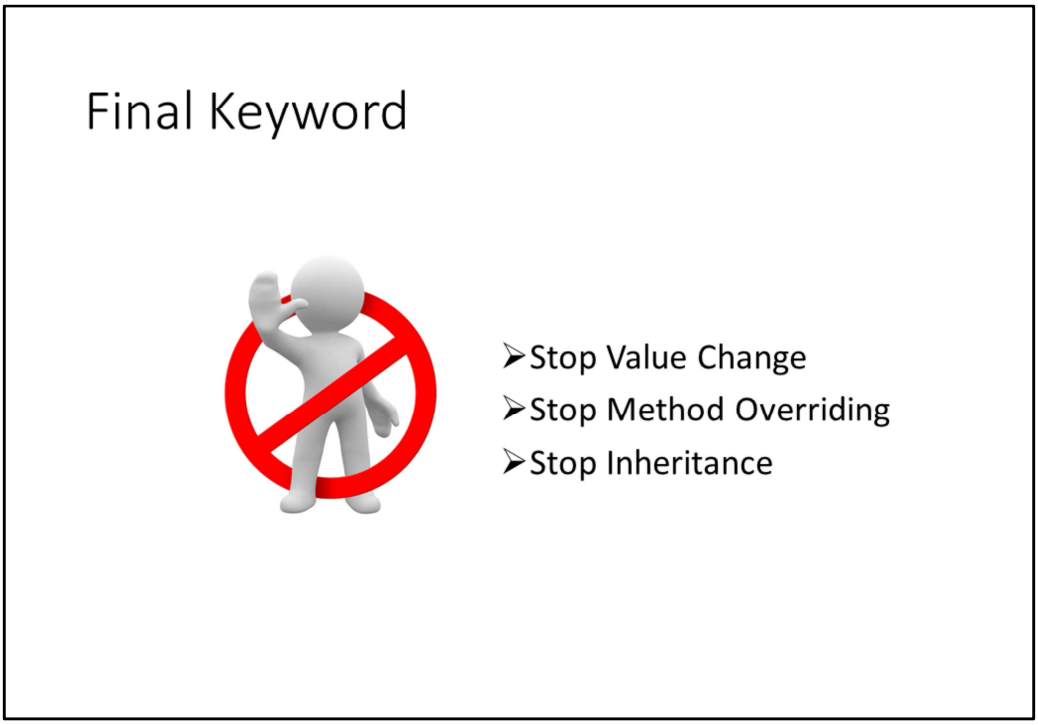
In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.



Final Keyword

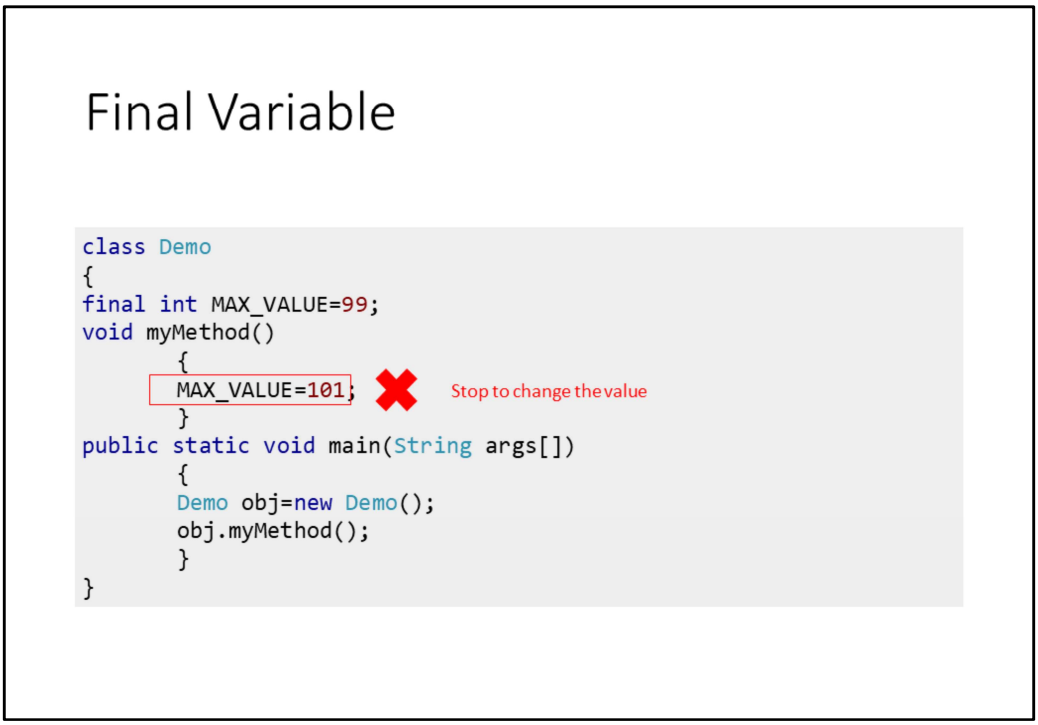
The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended



Final Variable

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Initializing a final variable

We must initialize a final variable, otherwise compiler will throw compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable :

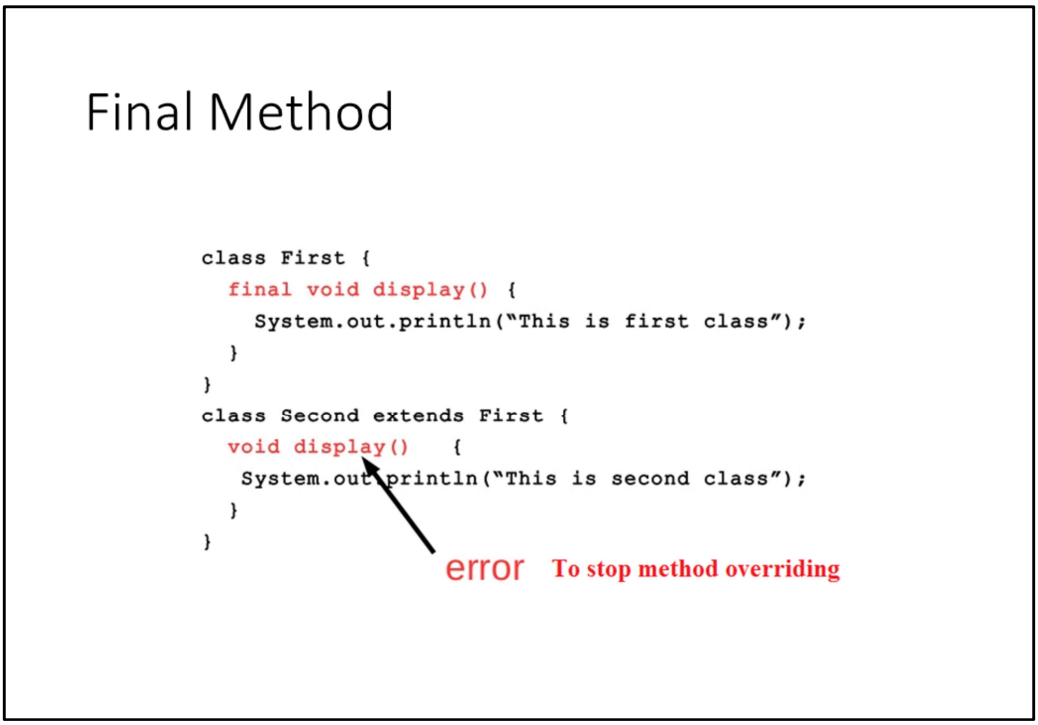
1. You can initialize a final variable when it is declared. This approach is the most common. A final variable is called **blank final variable**, if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside static block.

When to use a final variable :

The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Reference final variable

When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like *final StringBuffer sb;* As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity*.



Final Method

Methods marked as *final* cannot be overridden. When we design a class and feel that a method shouldn't be overridden, we can make this method *final*. We can also find many *final* methods in Java core libraries.

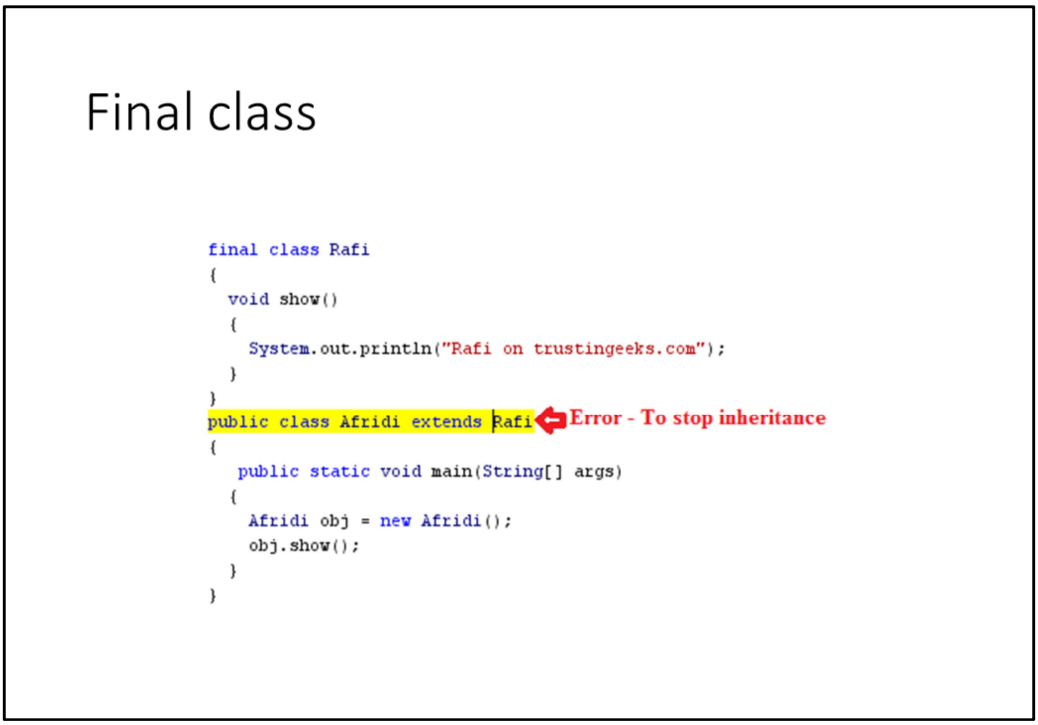
Sometimes we don't need to prohibit a class extension entirely, but only prevent overriding of some methods. A good example of this is the *Thread* class. It's legal to extend it and thus create a custom thread class. But its *isAlive()* methods is *final*.

This method checks if a thread is alive. It's impossible to override the *isAlive()* method correctly for many reasons. One of them is that this method is native. Native code is implemented in another programming language and is often specific to the operating system and hardware it's running on.

If some methods of our class are called by other methods, we should consider making the called methods *final*. Otherwise, overriding them can affect the work of callers and cause surprising results.

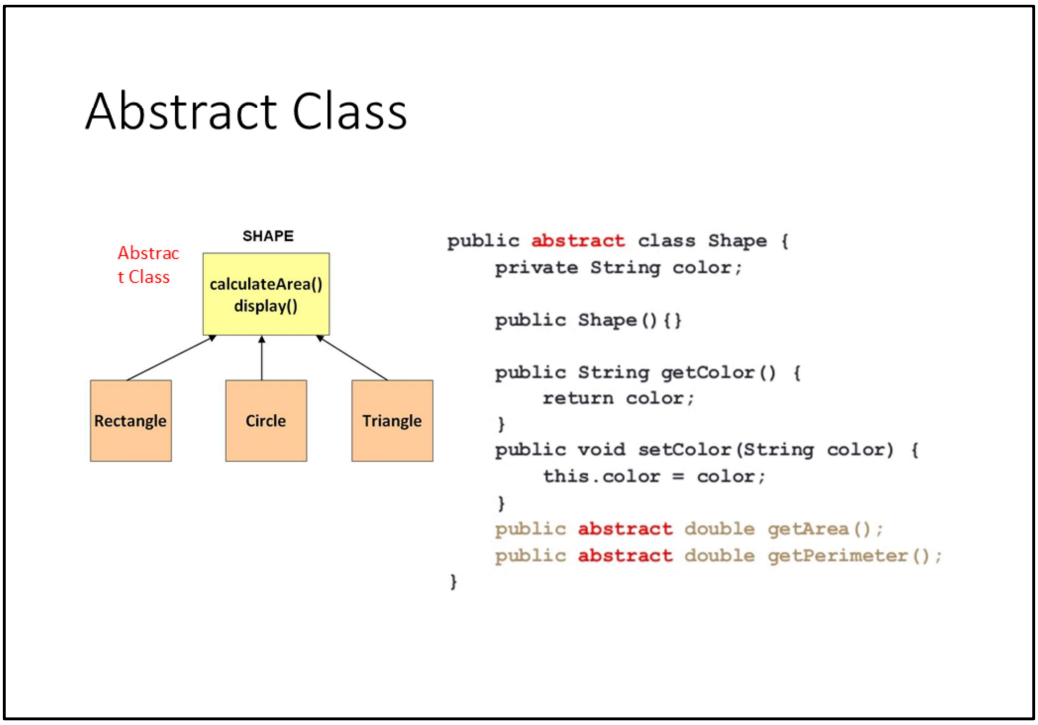
If our constructor calls other methods, we should generally declare these methods *final* for the above reason.

What's the difference between making all methods of the class *final* and marking the class itself *final*? In the first case, we can extend the class and add new methods to it.



Final Class

- **Classes marked as *final* can't be extended.** If we look at the code of Java core libraries, we'll find many *final* classes there. One example is the *String* class.
- Consider the situation if we can extend the *String* class, override any of its methods, and substitute all the *String* instances with the instances of our specific *String* subclass.
- The result of the operations over *String* objects will then become unpredictable. And given that the *String* class is used everywhere, it's unacceptable. That's why the *String* class is marked as *final*.
- Any attempt to inherit from a *final* class will cause a compiler error.
- Note that **the *final* keyword in a class declaration doesn't mean that the objects of this class are immutable**. We can change the fields of *class* object freely.
- We just can't extend it.
- If we follow the rules of good design strictly, we should create and document a class carefully or declare it *final* for safety reasons. However, we should use caution when creating *final* classes.
- Notice that making a class *final* means that no other programmer can improve it. Imagine that we're using a class and don't have the source code for it, and there's a problem with one method.
- If the class is *final,* we can't extend it to override the method and fix the problem. In other words, we lose extensibility, one of the benefits of object-oriented programming.



Abstract Class

There are many cases when implementing a contract where we want to postpone some parts of the implementation to be completed later. We can easily accomplish this in Java through abstract classes.

Characteristics:

- **We define an abstract class with the *abstract* modifier preceding the *class* keyword**
- An abstract class can be subclassed, but it **can't be instantiated**
- If a class defines one or more *abstract* methods, then the class itself must be declared *abstract*
- **An abstract class can declare both abstract and concrete methods**
- A subclass derived from an abstract class must either implement all the base class's abstract methods or be abstract itself

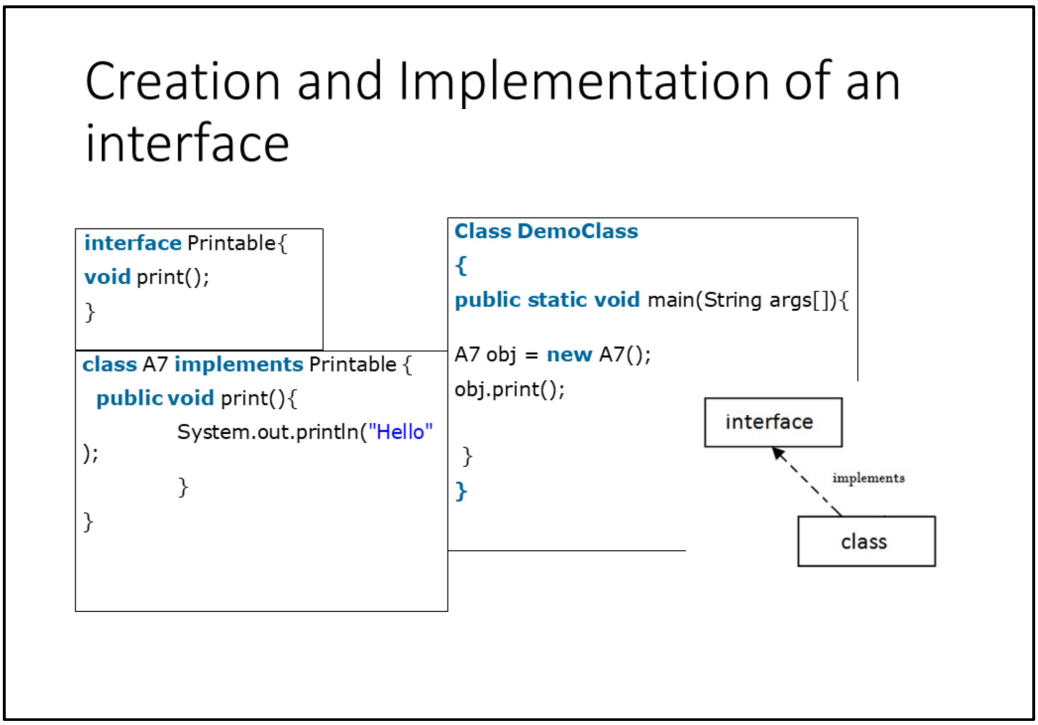
When to Use Abstract Classes

Now, **let's analyze a few typical scenarios where we should prefer abstract classes over interfaces** and concrete classes:

- We want to encapsulate some common functionality in one place (code reuse) that multiple, related subclasses will share
- We need to partially define an API that our subclasses can easily extend and refine
- The subclasses need to inherit one or more common methods or fields with protected access modifiers

Moreover, since the use of abstract classes implicitly deals with base types and subtypes, we're also taking advantage of Polymorphism.

Note that code reuse is a very compelling reason to use abstract classes, as long as the “is-a” relationship within the class hierarchy is preserved.



What Are Interfaces in Java?

In Java, an interface is an abstract type that contains a collection of methods and constant variables. It is one of the core concepts in Java and is **used to achieve abstraction, polymorphism and multiple inheritance.**

We can implement an interface in a Java class by using the *implements* keyword.

Rules for Creating Interfaces

In an interface, we're allowed to use:

1. **constants variables**
2. **abstract methods**
3. **static methods**
4. **default methods**

We also should remember that:

- we can't instantiate interfaces directly
- an interface can be empty, with no methods or variables in it
- we can't use the *final* word in the interface definition, as it will result in a compiler error
- all interface declarations should have the *public* or default access modifier; the *abstract* modifier will be added automatically by the compiler
- an interface method can't be *private*, *protected*, or *final*
- interface variables are *public*, *static*, and *final* by definition; we're not allowed to change their visibility

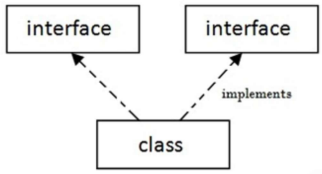
Multiple Inheritance using Interface

```
interface Printable{
void print();
}
```

```
interface  Showable{
  void show();
}
```

```
class A7 implements Printable,Showable{
  public void print(){
          System.out.println("Hello");
          }
public void show(){
          System.out.println("Welcome");
          }
}
```

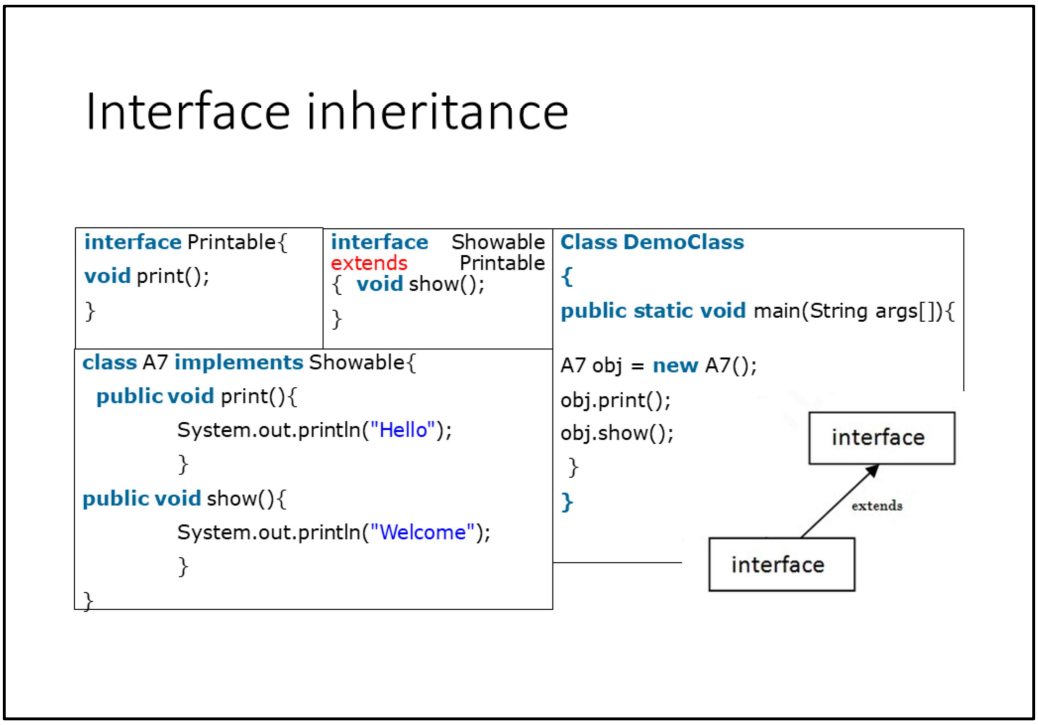
```
Class DemoClass
{
public static void main(String args[]){

A7 obj = new A7();
obj.print();
obj.show();
 }
}
```



One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. In addition, you can instantiate a class to create an object, which you cannot do with interfaces. An object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state*, which is the ability to inherit fields from multiple classes. For example, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence? Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple inheritance of implementation is the ability to inherit method definitions from multiple classes. Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass. Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.

- The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface.
- An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface.
- As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.



Interfaces Extending Other Interfaces

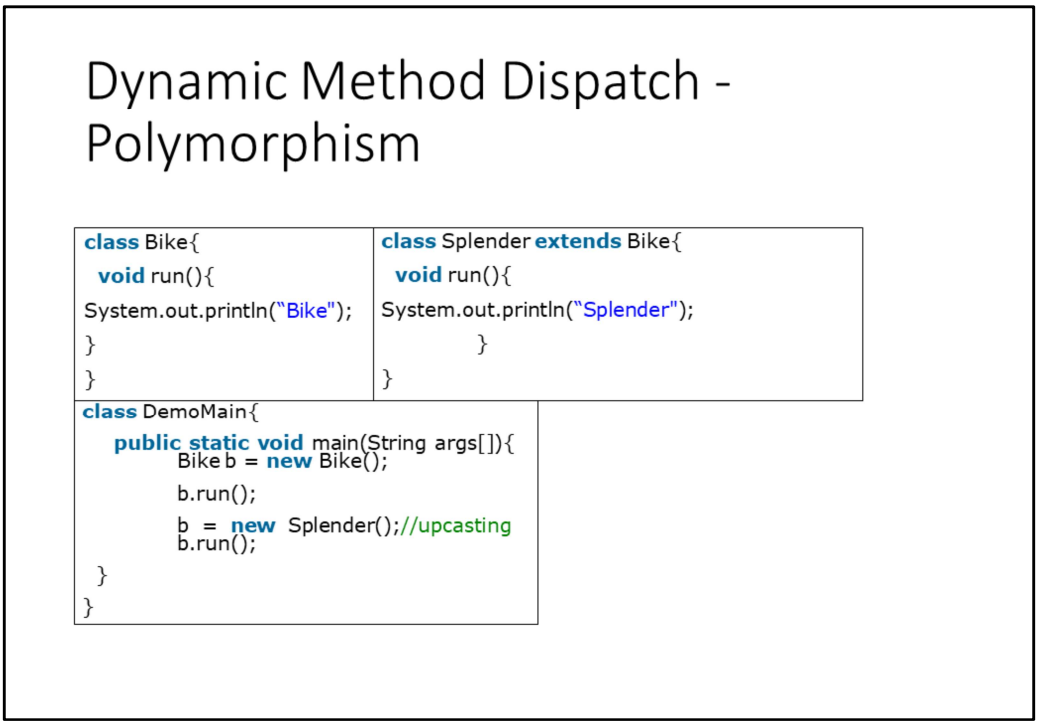
An interface inherits other interfaces by using the keyword *extends*. Classes use the keyword *implements* to inherit an interface.

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

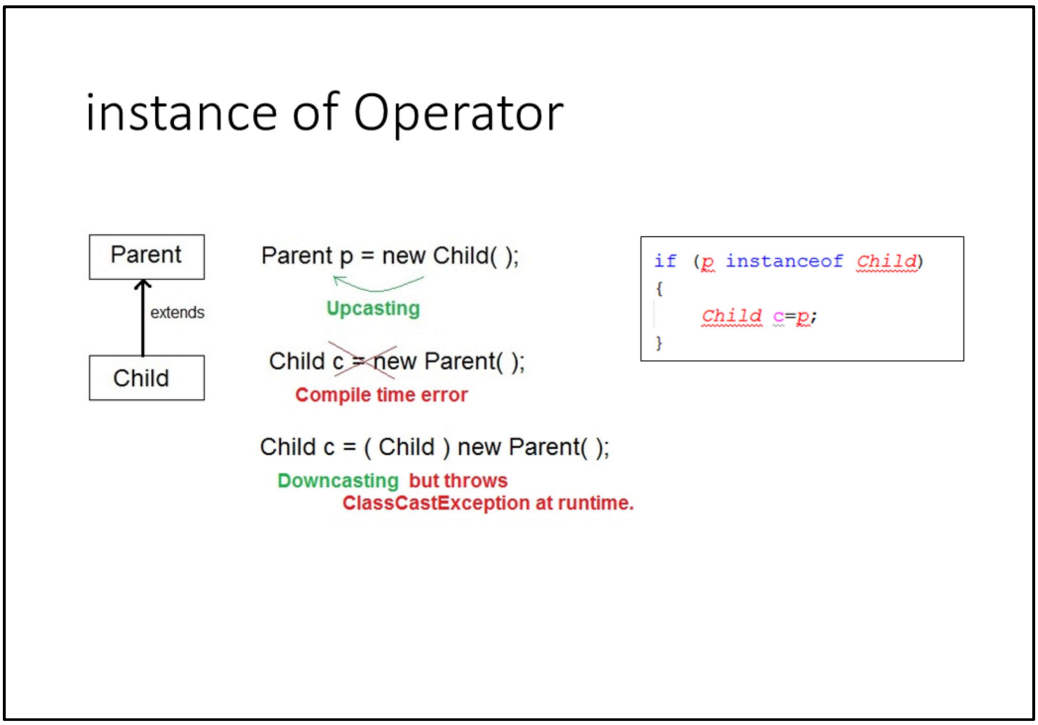
The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.



Dynamic Polymorphism

With dynamic polymorphism, the **Java Virtual Machine (JVM) handles the detection of the appropriate method to execute when a subclass is assigned to its parent form**. This is necessary because the subclass may override some or all of the methods defined in the parent class.

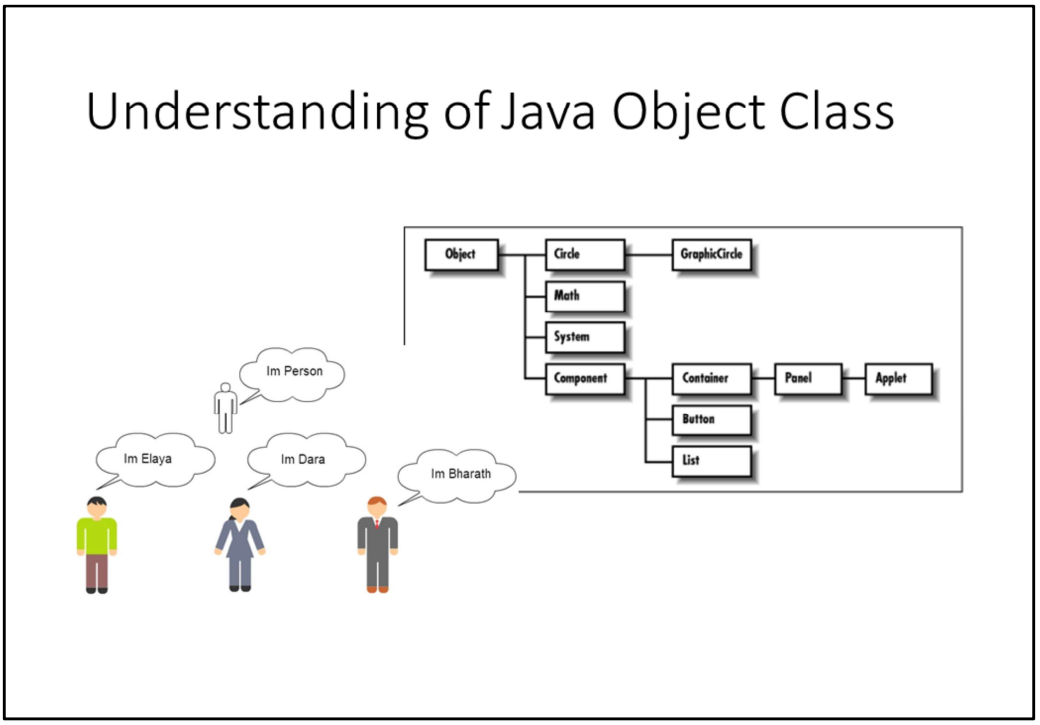
- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
- It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
- The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
- A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.



***instanceof* Operator**

instanceof is* a binary operator used to test if an object is of a given type. The result of the operation is either *true* or *false. It's also known as type comparison operator because it compares the instance with type.

- Before casting an unknown object, the *instanceof* check should always be used. Doing this helps in avoiding *ClassCastException* at runtime.
- The instanceof operator works on the principle of the is-a relationship. The concept of an is-a relationship is based on class inheritance or interface implementation.
- If we use the instanceof operator on any object that is null, it returns false. Also, no null check is needed when using an instanceof operator.
- Instance tests and casts depend on inspecting the type information at runtime. Therefore, we can't use instanceof along with erased generic types.



Object class

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
The Object class is beneficial if you want to refer any object whose type you don't know.
Notice that parent class reference variable can refer the child class object, know as upcasting.

Methods of Object class

- public final Class getClass() -> returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
- public int hashCode() -> returns the hashcode number for this object.
- public boolean equals(Object obj) -> compares the given object to this object.
- protected Object clone() throws CloneNotSupportedException -> creates and returns the exact copy (clone) of this object.
- public String toString() -> returns the string representation of this object.
- public final void notify() -> wakes up single thread, waiting on this object's monitor.
- public final void notifyAll() -> wakes up all the threads, waiting on this object's monitor.
- public final void wait(long timeout)throws InterruptedExceptionc -> auses the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
- public final void wait(long timeout,int nanos)throws InterruptedException -> causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
- public final void wait()throws InterruptedException -> causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
- protected void finalize()throws Throwable -> is invoked by the garbage collector before object is being garbage collected.

Abstract vs Interface

Feature	Interface	Abstract Class
Multiple Inheritance	Yes	No
Object Instantiated	No	No
Instance Member Function (Methods)	No	Yes
Instance Data Member	No	Yes
Inheritance	Implements	Extends
Access Modifier	Public	All
Constructor	No	Yes

Abstract class vs Interface

- **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
- **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
- **Type of variables:** Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
- **Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
- **Inheritance vs Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
- **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

References:

- http://www.write-technical.com/126581/session7/session7.htm
- http://sourcecodemania.com/inheritance-in-java/
- http://www.sitesbay.com/java/java-method-overloading

Questions/Comments