Core Java

Inheritance & Polymorphism

Objective

At the end of this session, you will be able to:

- Use Wrapper Classes
- Write programs to implement Inheritance
- Write programs using method Overloading and Overriding
- Write Abstract classes & Interfaces

Agenda

- Wrapper Classes
- Inheritance
  - Introduction
  - Types of Inheritance
  - Object class and its methods
- Polymorphism
  - Method Overloading and Overriding
  - Static Polymorphism and Dynamic Polymorphism
- Abstract class
- Interface

Wrapper Classes

- Java treats objects differently from variables of Primitive types

  Some times we need to treat int, char, float values as Objects

  Java provides Wrapper Classes for each primitive type which wraps the value as an Object

  Converting from primitive to wrapper class is called as Boxing

```
Integer intobj = new Integer(575);
```

  Converting from wrapper class to primitive is called as Unboxing

```
int i = intobj.intValue();
```

Wrapper Classes (Contd…)

Primitive Type	Wrapper Class	Constructor Arguments
byte	Byte	byte or String
short	Short	short or String
int	Integer	int or String
long	Long	long or String
float	Float	float, double or String
double	Double	double or String
char	Character	char
boolean	Boolean	boolean or String

Note: The Wrapper classes do not contain a no-argument constructor

Methods in Number class

- parseXXX(String)
- toString()
- valueOf(String)

Method Summary

byte	**byteValue**() Returns the value of the specified number as a `byte`.
abstract double	**doubleValue**() Returns the value of the specified number as a `double`.
abstract float	**floatValue**() Returns the value of the specified number as a `float`.
abstract int	**intValue**() Returns the value of the specified number as an `int`.
abstract long	**longValue**() Returns the value of the specified number as a `long`.
short	**shortValue**() Returns the value of the specified number as a `short`.

Fields in Number classes

Field Summary

static long	**MAX_VALUE** A constant holding the maximum value a `long` can have, $2^{63}-1$.
static long	**MIN_VALUE** A constant holding the minimum value a `long` can have, -2^{63}.
static int	**SIZE** The number of bits used to represent a `long` value in two's complement binary form.

Wrapper Classes (Contd…)

- Wrapper classes have a lot of useful methods

  Examples:

  ```
  Character.toLowerCase(ch)
  Character.isLetter(ch)
  ```

- A common translation is converting a string to a numeric type such as an int

  Example:

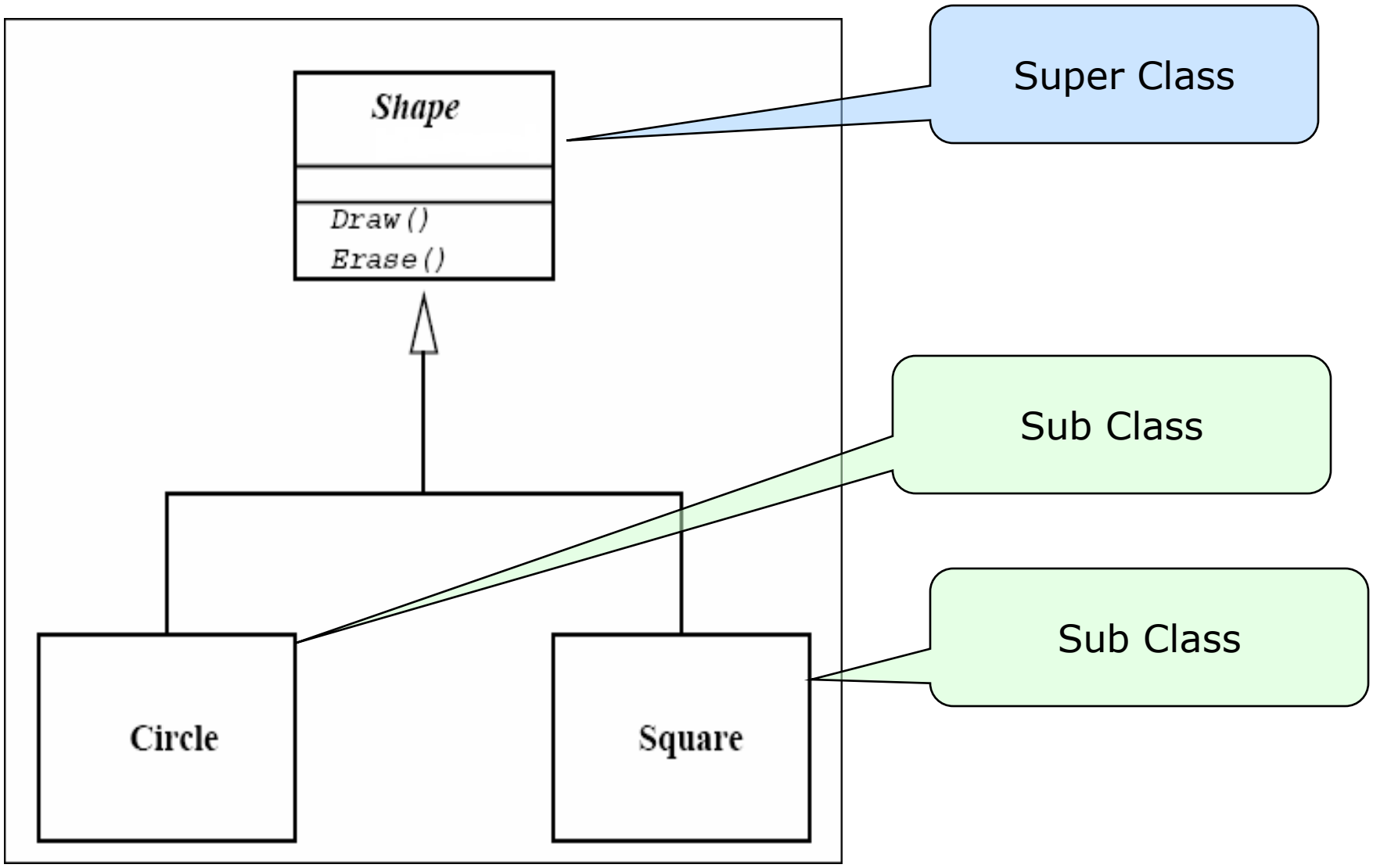
  ```
  String s = "65000";
  int i = Integer.parseInt(s);
  ```

WrapperDemo.java

Inheritance

- Implements IS-A relationship
- Capability of a class to use the properties & methods of another class while adding its own functionality
- A class derived from another class is called as subclass / / derived class / extended class / child class
- The class from which the subclass is derived is called as superclass / base class / parent class
- Each class is allowed to have one direct superclass
- Each superclass can have unlimited number of subclasses

Inheritance (Contd…)



Inheritance (Contd…)

- A class is declared subclass of another class by using the extends keyword

```
class SubClass extends BaseClass{
    …
}
```

- All base class fields & methods are inherited by the subclass

  private members of the super class are not accessible directly by any method of the subclass

  static data members of the base class are also inherited

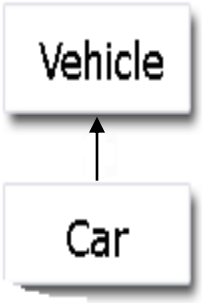
  protected members of the super class are inherited by subclass

TestInheritance.java

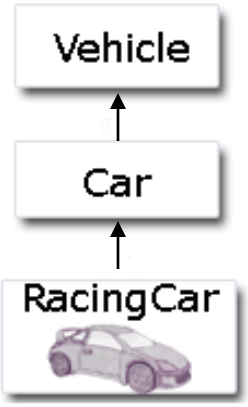
Types of Inheritance

- Java supports two types of inheritance

Single Inheritance



Multilevel Inheritance



Note: Java doesn't support Multiple Inheritance
(One sub class deriving from more than one super classes)

Constructors in Inheritance

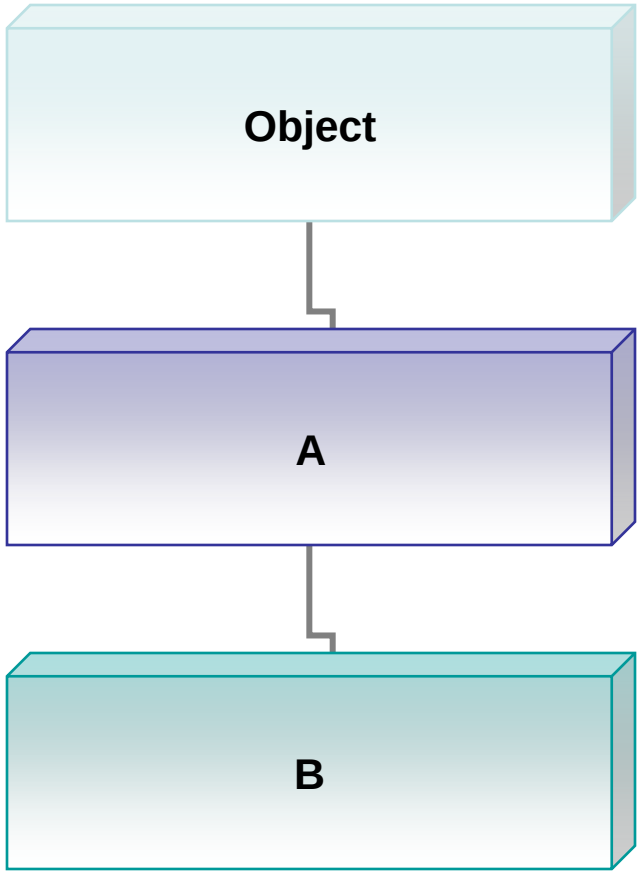
- Constructors are invoked in the order of hierarchy
- While instantiating a sub class, its super class default constructor will be invoked first, followed by the sub class constructor
- The keyword super can be used to invoke the super class parameterized constructor instead of the default
- Remember that:
  - super() call must occur as the first statement in constructor
  - super() call can only be used in a constructor definition

MutiInheritanceDemo.java

The Object Class

- Object is the base class for all Java classes
- Every class extends this class directly or indirectly
- Present in the package java.lang which is imported by default into all java programs

```
class A
{
        …
}
public class B extends A
{
        …
}
```



Methods of Object Class

toString()	Returns a string representation of the object
finalize()	Called by the garbage collector on an object when there are no more references to the object
equals()	Indicates if some other object is "equal to" this one
clone()	Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object
hashCode()	Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable

ObjectDemo.java

Give this a Try…

1. Observe the following snippet and think:

```
 class Animal
{
        …
}
public class Cat extends
Animal
{
        …
}
```

2. Class Cat has access to all the methods and variables declared in class Animal. State True / False

3. I can declare a few members in class Cat. State True / False

4. Name the super classes extended by the class Cat directly or indirectly

Polymorphism

- Means one entity having many (poly) forms (morph)
- We can have multiple methods with the same name in the same class, i.e. Method Overloading
- Capability of an action or *method* to do different things based on the object that it is acting upon, i.e. Method Overriding

Method Overloading

- Two methods in a class can have same name, provided their signature is different

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) {
        System.out.println("double d = " + d);
    }
}
```

Same name with different parameters

OverloadDemo.java

Method Overriding (Contd…)

- Useful if a derived class needs to have a different implementation of a certain method from that of the superclass
- A subclass can override a method defined in its superclass by providing a new implementation for that method
- The new method definition must have the same method signature (i.e., method name & parameters) and return type
- The new method definition cannot narrow the accessibility of the method, but it can widen it

Using *super*

- The keyword super refers to members of the super class
- *U*sed when member names of the subclass hide members by the same name in the super class
- super.member

  member can be either a method or an instance variable

SuperDemo.java

Using *final*

- Using *final* to prevent Overrriding

  When a method in the superclass is declared as final, it cannot be overridden in the subclass

- Using *final* to prevent Inheritance

  When a class is declared as final, it cannot be inherited

Give this a Try…

1. Which of the following are valid examples of Polymorphism?

    a. public void add(String s,int i);
    b. public int add(String s,int i);
    c. public void add(String s,int i,int j);
    d. public void add(int I,String s);

2. Inheriting final class causes run time exception — State True or False

Type Casting for Reference Types

- We can assign a variable of a certain class type with an instance of that particular class or an instance of any subclass of that class

```
public class Mammal { .. . . .  }
public class Dog extends Mammal {  . . . .  }
```

- When we cast a reference along the class hierarchy in a direction from the root class towards the subclasses, it's a downcast

```
Mammal m=new Mammal();
Dog d=(Dog) m; // explicit casting
```

- When we cast a reference along the class hierarchy in a direction from the sub classes towards the root, it's an upcast

```
Mammal m=new Dog();// implicit casting
```

Binding

- Happens when a method invocation is bound to an implementation
  - Involves lookup of the method in the class, or one of its parents
  - Both method names & parameters are checked
- Can happen at :
  - Compilation Time (Static Binding)
  - Execution Time (Dynamic Binding)

Static Binding

- Static binding is done by the compiler
  - When it can determine the type of an object
- Method calls are bound to their implementation immediately

```
    public class MotorBike
    {
        public void revEngine() {…}
    }
MotorBike bike = new MotorBike();
bike.revEngine();
```

Dynamic Binding

- When an object's class cannot be determined at compile time
- JVM (not the compiler) has to bind a method call to its implementation
- Instances of a sub-class can be treated as instances of the parent class
- So the compiler doesn't know its type, just knows its base type

```
public class Mammal { .. . . .  }
public class Dog extends Mammal {  . . . .  }
public class Cat extends Mammal {  . . . .  }
Mammal m;
m = new Dog( );
m.speak(); // Invokes Dog's implementation
m= new Cat();
m.speak(); // Invokes Cat's implementation
```

Abstract Class

- A class that is declared with *abstract* keyword
- May or may not include abstract methods
  - Abstract methods do not have implementation (body)
- Cannot be instantiated, but it can be subclassed
- When an abstract class is subclassed, the subclass provides implementations for all abstract methods in its parent class
- And, if it does not, the subclass must also be declared abstract

Abstract Class (Contd…)

```
abstract class Shape
{
    public abstract void draw();
}
```

Concrete class

A class can extend only one class

```
class Circle extends Shape
{
    public void draw()
    {
            -------
    }
}
```

TestAbstract.java

Interface

- In Java, an *interface* is a reference type, similar to a class
- Can contain constants & method signatures
- There are no method bodies
- Cannot be instantiated — they can only be *implemented* by classes or *extended* by other interfaces
- Variables & methods declared in interfaces are public by default
- An interface can extend another interface
- A class can implement multiple interfaces

Interface (Contd…)

Why Interface?

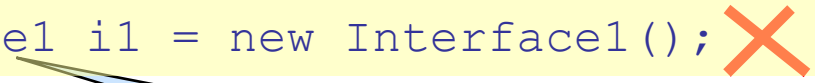
- To reveal an object's programming interface (functionality of the object) without revealing its implementation
- To have unrelated classes implement similar methods (behaviors)
- To model multiple inheritance

Interface (Contd…)

Example of an interface declaration:

```
public interface Interface1
{
    public void set(int i);
    public int get();
}
```

```
Interface1 i1 = new Interface1();
```



Interfaces and Abstract classes can not be instantiated

TestInterface.java

The *instanceOf* Operator

- Used to check the actual type of an object
- Has two operands:
  - A reference to an object on the left
  - A class name on the right
- Returns true or false based on whether the object is an instance of the named class or any of that class's subclasses

The *instanceOf* Operator (Contd…)

An Example:

A class definition

```
public class MotorBike implements Vehicle, Motorised
{
        …
}
```

```
 Vehicle bike = new MotorBike();
 if (bike instanceOf MotorBike)
  {
        -------
 }
```

Check whether the bike reference is an instance of MotorBike or not

Summary

In this session, we have covered:

- Wrapper Classes
- Inheritance
- Polymorphism
- Abstract Class
- Interface

Thank You