

Dynamic Method Dispatch

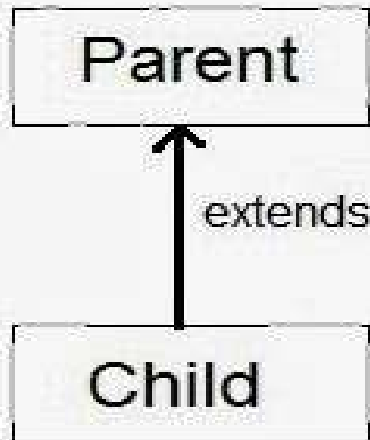
Abstract Class



Prepared by

Dr. Rajesh Kumar Ojha
Asst. Prof., CSE, Silicon University

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- This is how Java implements run-time polymorphism.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.



```
Parent p = new Parent( );
```

```
Child c = new Child( );
```

```
Parent p = new Child( );
```

Upcasting

```
Child c = new Parent( );
```

incompatible type

```
class Figure {  
    double dim1;  
    double dim2;  
  
    Figure(double a, double b) {  
        dim1 = a;  
        dim2 = b;  
    }  
  
    double area() {  
        System.out.println("Area for Figure is undefined.");  
        return 0;  
    }  
}
```

```
class Rectangle extends Figure {  
    Rectangle(double a, double b) {  
        super(a, b);  
    }  
  
    // override area for rectangle  
    double area() {  
        System.out.println("Inside Area for Rectangle.");  
        return dim1 * dim2;  
    }  
}
```

```
class Triangle extends Figure {  
    Triangle(double a, double b) {  
        super(a, b);  
    }  
  
    // override area for right triangle  
    double area() {  
        System.out.println("Inside Area for Triangle.");  
        return dim1 * dim2 / 2;  
    }  
}
```

```
class FindAreas {  
    public static void main(String args[]) {  
        Figure f = new Figure(10, 10);  
        Rectangle r = new Rectangle(9, 5);  
        Triangle t = new Triangle(10, 8);  
  
        Figure figref;  
  
        figref = r;  
        System.out.println("Area is " + figref.area());  
  
        figref = t;  
        System.out.println("Area is " + figref.area());  
  
        figref = f;  
        System.out.println("Area is " + figref.area());  
    }  
}
```

```
class A
{
void callme()
{
System.out.println("Inside A's callme method");
}
}
class B extends A
{
void callme() // override callme()
{
System.out.println("Inside B's callme method");
}
}
class C extends A
{
void callme() // override callme()
{
System.out.println("Inside C's callme method");
}
}
```



```

public class Dynamic_disp
{
public static void main(String args[])
{
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}

```

Inside A's callme method
Inside B's callme method
Inside C's callme method

- Here reference of type A, called r, is declared.
- The program then assigns a reference to each type of object to r and uses that reference to invoke callme().
- As the output shows, the version of callme() executed is determined by the type of object being referred to at the time of the call.

ABSTRACT CLASS

- If a class contain any abstract method then the class is declared as abstract class.
- An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method.

Syntax:

```
abstract class class_name { }
```

- Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
- An abstract class must have an abstract method.
- Abstract classes can have Constructors, Member variables and Normal methods.
- Abstract classes are never instantiated.
- When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

ABSTRACT METHOD

- Method that are declared without any body within an abstract class is known as abstract method.
- The method body will be defined by its subclass.
- Abstract method can never be final and static.
- Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax:

`abstract return_type function_name (); // No
definition`

Example

```
abstract class A {  
    abstract void callme();  
}  
class B extends A {  
    void callme() {  
        System.out.println("this is callme.");  
    }  
    public static void main(String[] args) {  
        B b=new B(); b.callme();  
    }  
}
```

output: this is callme.

Abstract class with concrete(normal) method

```
abstract class A {  
    abstract void callme();  
    public void normal() {  
        System.out.println("this is concrete method");  
    }  
}  
  
class B extends A {  
    void callme() {  
        System.out.println("this is callme.");  
    }  
  
    public static void main(String[] args) {  
        B b=new B(); b.callme(); b.normal();  
    }  
}
```

Abstraction using abstract class

- Abstraction is an important feature of OOPS.
- It means hiding complexity.
- Abstract class is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method. Lets see how abstract class is used to provide abstraction.

```
abstract class Vehicle {  
    public abstract void engine();  
}  
  
public class Car extends Vehicle {  
    public void engine() {  
        System.out.println("Car engine"); // car engine implementation  
    }  
  
    public static void main(String[] args) {  
        Vehicle v = new Car();  
        v.engine();  
    }  
}
```

Output: Car engine

- Here by casting instance of **Car** type to **Vehicle** reference, we are hiding the complexity of **Car** type under **Vehicle**.
- Now the **Vehicle** reference can be used to provide the implementation but it will hide the actual implementation process.

When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations.
- These subclasses extend the same Abstract class and provide different implementations for the abstract methods.
- Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

```

abstract class Shape {
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape {
void draw() {System.out.println("drawing rectangle");}
}
class Circle1 extends Shape {
void draw() {System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1 {
public static void main(String args[]) {
Shape s=new Circle1(); //In real scenario, object is provided through method e.g. getShape() method
s.draw();
}
}
Output: drawing circle

```

Thank you