# Lecture – 20

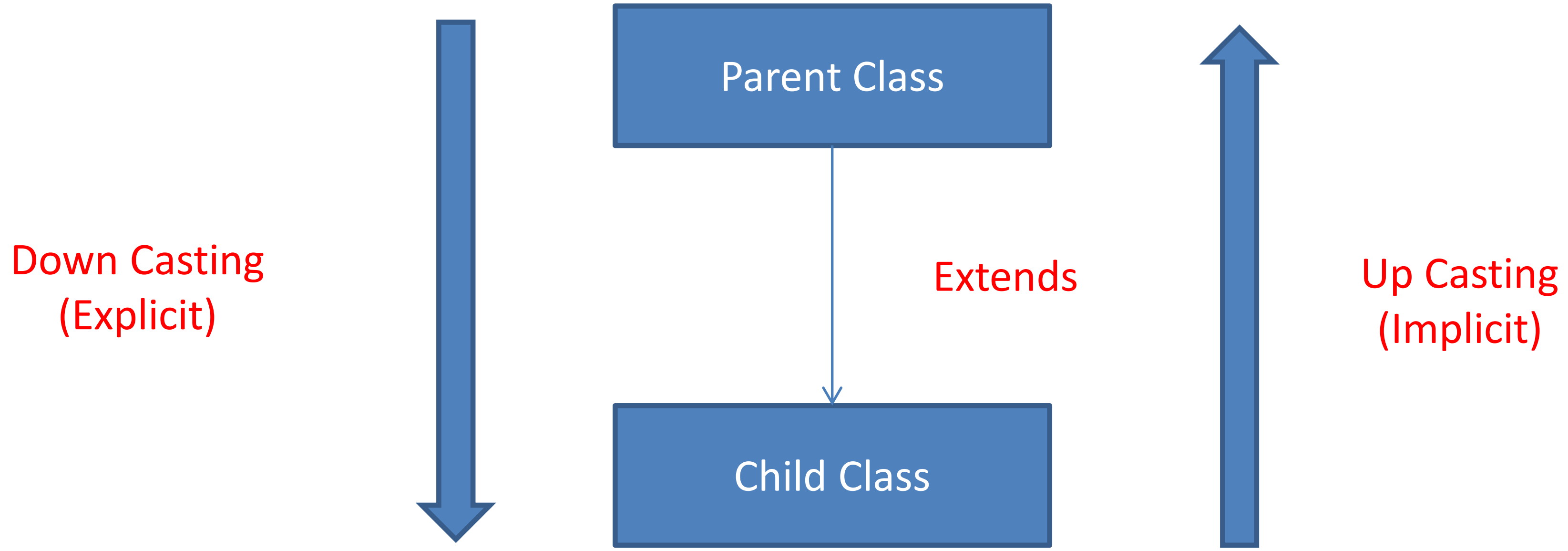
UpCasting and DownCasting

## Type Casting

- Type casting is a process in which we change the type of one data type to another data type
- In java, we can type cast Primitive Variables and Reference Variables
- Type casting of reference variables is also known as Reference Type Casting
- Reference Variables contain the reference of the objects
- Objects can never be casted into other type of objects

# Types of Reference Type Casting

- Two types of Reference Type Casting
  - Up Casting
  - Down Casting



For Type Casting there must be ISA relationship

# Up Casting (Implicit)

- Syntax

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

```
Parent obj = new Child();    =    Child cobj = new Child();
                                  Parent obj = (Parent)cobj;
```

- It is used in run-time polymorphism
- When doing implicit casting, the instance/overloaded/static methods or variables are accessed based on the reference type.
- **Parent p** = new Child();        Parent p belongs to the Parent Class

# Up Casting - Methods

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

| **Aspect** | **Instance Methods** | **Static Methods** | **Overloaded Methods** |
|---|---|---|---|
| Behavior | Resolved at runtime based on object type. Polymorphism applies. Dynamic method dispatch applies. | Resolved at compile-time based on reference type. | Resolved at compile-time based on reference type. |
| Access | Based on object type at runtime. | Based on reference type at compile-time. | Based on reference type at compile-time. |
| Overriding | Can be overridden by subclasses. Dynamic polymorphism applies. | Cannot be overridden. | Overloaded methods have different signatures but can be in the same class or in different classes. |

## Up Casting (Implicit)

```
class Parent{
        int a = 10;
        void parentMethod(){
                System.out.println("Inside Parent Class");
        }
        int sum(int a, int b){return a+b;}
}

class Child extends Parent{
        int b = 20;
        void childMethod(){
                System.out.println("Inside Child Class");
        }
        double sum(double a, double b){return a+b;}

}
```

```
public class Runner{
        public static void main(String[] args){
                Parent p = new Child();

        System.out.println(p.b);//Error
        Parent p = new Child();
                p.parentMethod();

        System.out.println(p.sum(2,3));         }
}
```

# Up Casting (Implicit)

- In case of run time polymorphism
  - The overridden methods are picked based on the child object

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

```
class Parent{
		void method(){
				System.out.println("Inside Parent Class");
		}
}
class Child extends Parent{
		void method(){
				System.out.println("Inside Child Class");
		}
}
```

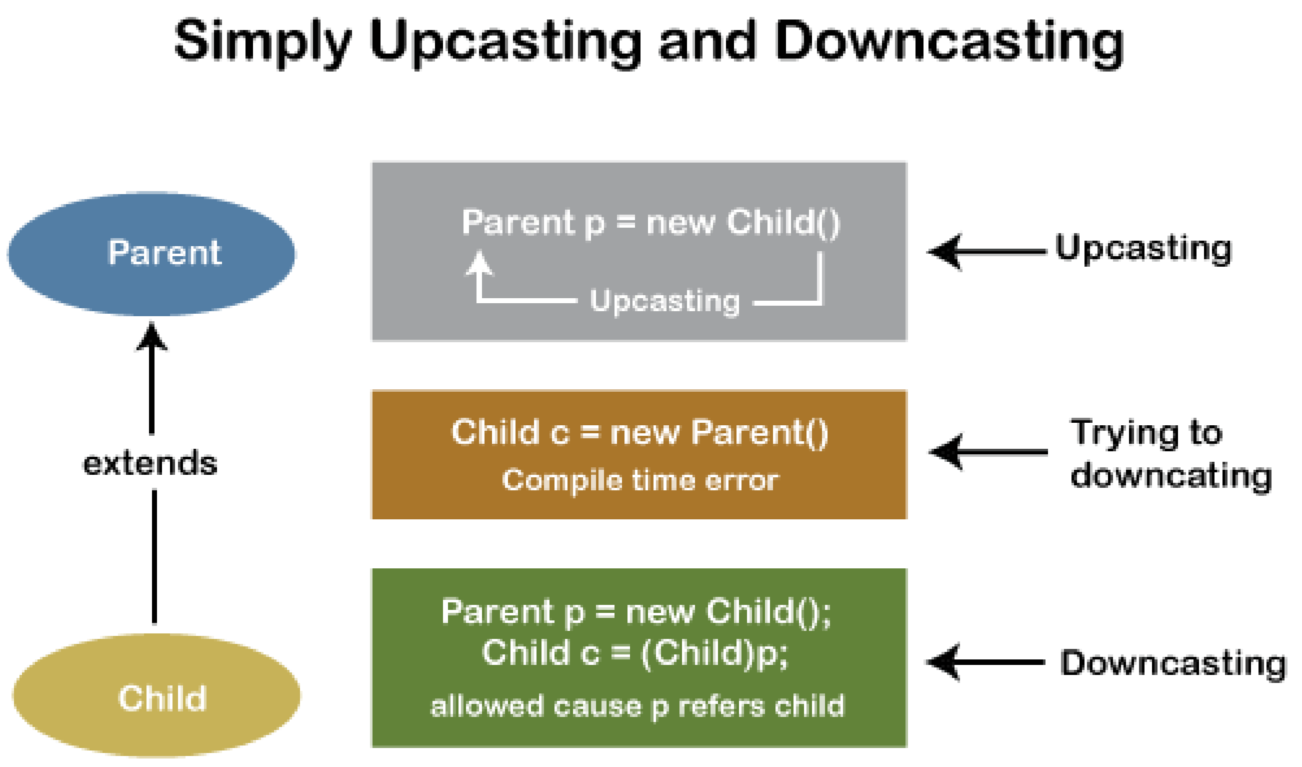
```
public class Runner{
		public static void main(String[] args){
				Parent p = new Child();
				p.method();
		}
}
```

# Down Casting

- Casting parent class reference to child class.
- First do up casting then perform down casting
- Does not work implicitly
  - Child obj = new Parent(); //Error;  we can not say Parent ISA Child
  - Child obj = (Child) new Parent(); // compiles but ClassCastException will thrown
- Syntax

```
ParentClass obj = new ChildClass();
((ChildClass)obj) OR
ChildClass cObj =  (ChildClass)obj;
```

# Up Casting and Down Casting



## *instanceof* Operator

- Used to check the object is an instance of the specified type (class or subclass or interface)
  - Return true or false
- Also known as type *comparison operator* because it compares the instance with type
- ***instanceof*** operator will return false if used with any variable that has null value.
- Syntax

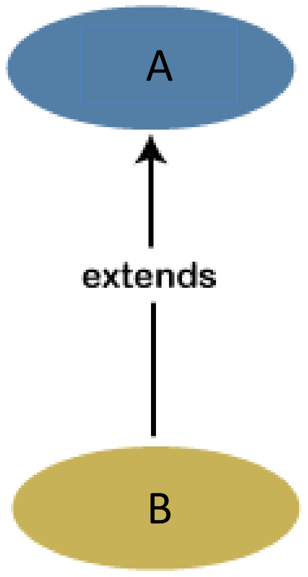
refVariable ***instanceof*** ClassName

a ***instanceof*** A

# *instanceof* Operator

- Better to check before down cast

```
A a = new B();
if(a instanceof B){
    B b = (B)a;
    System.out.println("Down Cast");
}
```



**A a = new A();**

```
System.out.println (a instanceof A);    //true
System.out.println (a instanceof Object);  //true
System.out.println (a instanceof B); //false
System.out.println (a instanceof String);// Error Incompatible types
```

**B a = new B();**

```
System.out.println (a instanceof A);    //true
System.out.println (a instanceof Object);  //true
System.out.println (a instanceof B); //true
```

**A a = new B();**

```
System.out.println (a instanceof A);    //true
System.out.println (a instanceof Object);  //true
System.out.println (a instanceof B); //true
```

# *Equals()* Method of Object Class

- *Like the* **toString()** *method, the* **equals(Object)** *method is another useful method defined in the* **Object** *class.*
- Signature: **public boolean** equals(Object o)
- This method tests whether two objects are equal

```
object1.equals(object2);
```

- The default implementation of the **equals** method in the **Object** class is

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

This implementation checks whether two reference variables point to the same object using the **==** operator

## *Equals()* Method of Object Class

- Correct way to override equals method

```
public boolean equals(Object o) {
        if (o instanceof Circle)
                return radius == ((Circle)o).radius;
        else
                return this == o;
}
```

## Example

```
public class Test {
        public static void main(String[] args) {
                Object circle1 = new Circle();
                Object circle2 = new Circle();
                System.out.println(circle1.equals(circle2));
        }
}
```

```
class Circle {
        double radius;
        public boolean equals(Circle circle) {
                return this.radius == circle.radius;
        }
}
```

The output is false if the Circle class in (a) is used. The Circle class has **two overloaded methods**: **equals(Circle circle)** defined in the Circle class and **equals(Object o)** defined in the Object class, inherited by the Circle class. At compile time, **circle1.equals(circle2) is matched to equals(Object o), because the declared type for circle1 and circle2 is Object**.

```
class Circle {
        double radius;
        public boolean equals(Object circle) {
                return this.radius == ((Circle)circle).radius;
        }
}
```

The output is true if the Circle class in (b) is used. The Circle class overrides the equals(Object o) method defined in the Object class. At compile time, circle1.equals(circle2) is matched to equals(Object o) and at runtime the equals(Object o) method implemented in the Circle class is invoked.