UNIT-IV

Inheritance, Packages & Interfaces

Abstract

- Method overriding
- Super keyword
- Dynamic method dispatch
- Object class

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java.**

- **Usage:**
  - Method overriding is used to provide the specific implementation of a method which is already provided by its super class.
  - Method overriding is used for **runtime polymorphism**

Method Overriding

- **Rules for Java Method Overriding**

- The method must have the **same name as in the parent class**
- The method must have the **same parameter** as in the parent class.
- There must be an **IS-A relationship (inheritance).**

Method Overriding

```
class Vehicle
{
    void run()
    {     System.out.println("Vehicle is running");
}
}
class Bike2 extends Vehicle
{
    void run()  // Method overriding
    {     System.out.println("Bike is running safely");
}

 public static void main(String args[])
{
    Bike2 obj = new Bike2();
    obj.run();
 }
}
```

Super keyword

- The **super** keyword in java is a **reference variable** which is used to **refer immediate parent class object.**

- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Super keyword

- **super to access superclass members**

- If your method overrides one of its superclass methods, you can invoke the **overridden** method through the use of the keyword **super**.

- **Example:**

Super keyword

```
class A
{
    String name = "Class A";
    public void display()
    {
        System.out.println("Class A display method called..");
    }
}
class B extends A
{
    String name = "Class B";
    public void display()
    {
        System.out.println("Class B display method called..");
    }
```

Super keyword

```
void printName()
{
	System.out.println("Name from subclass : " + name);
	System.out.println("Name from Superclass: " + super.name);
	display();
	super.display();
}
}
class SuperDemo
{
	public static void main(String args[])
	{
		B b1 = new B();
		b1.printName();
	}
}
```

Super keyword

- **super to call superclass constructor**

- Every time a parameterized or non-parameterized constructor of a subclass is created, then by default a default constructor of superclass is called implicitly.

- **Syntax:**

super();

OR

super(parameter list);

- **Example:**

Super keyword

```
class A // super class
{
    A() // default constructor
    {
    System.out.println("Super class default constructor called..");
    }
    A(String s1)  // parameterized constuctor
    {
    System.out.println("Super class parameterized constructor
   called: "+s1);
    }
}
```

Super keyword

```
class B extends A
{
    B()  // default constructor
    {
    System.out.println("Sub class default constructor called..");
    }
    B(String s1) // parameterized constructor
    {
    super("Class A");
    System.out.println("Sub class parameterized constructor
   called: " + s1);
}
}
```

Super keyword

```
class SuperConDemo
{
	public static void main(String args[])
	{
		B b1 = new B();
		B b2 = new B("Class B");
}
}
```

Dynamic method dispatch

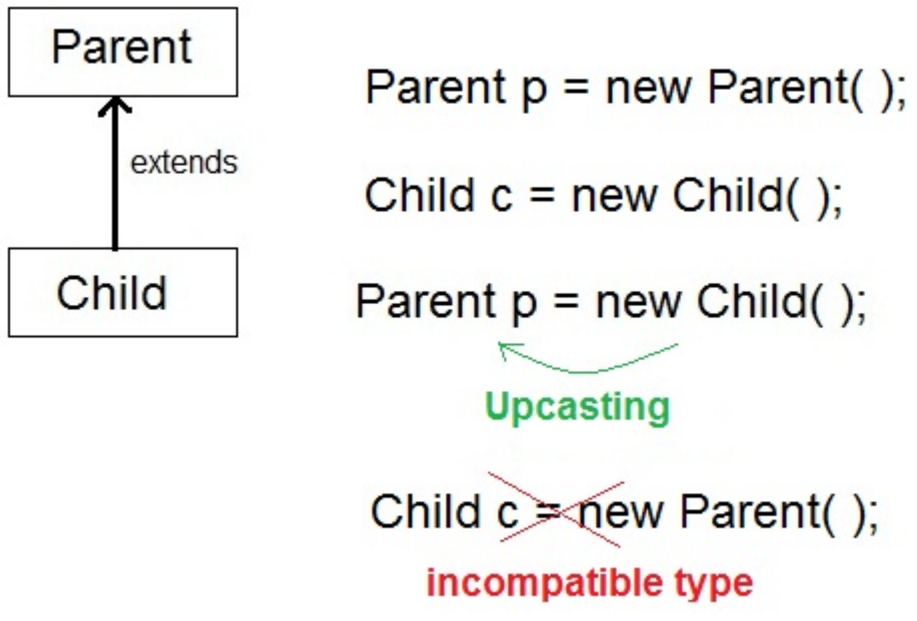
- Dynamic Method Dispatch is a process in which a **call to an overridden method is resolved at runtime** rather than compile-time.

- **Runtime polymorphism**

- In this process, an overridden method is called through the reference variable of a super class.

Dynamic method dispatch

- **Upcasting**
  - If the reference variable of Parent class refers to the object of Child class, it is known as **upcasting**.



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

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

Dynamic method dispatch

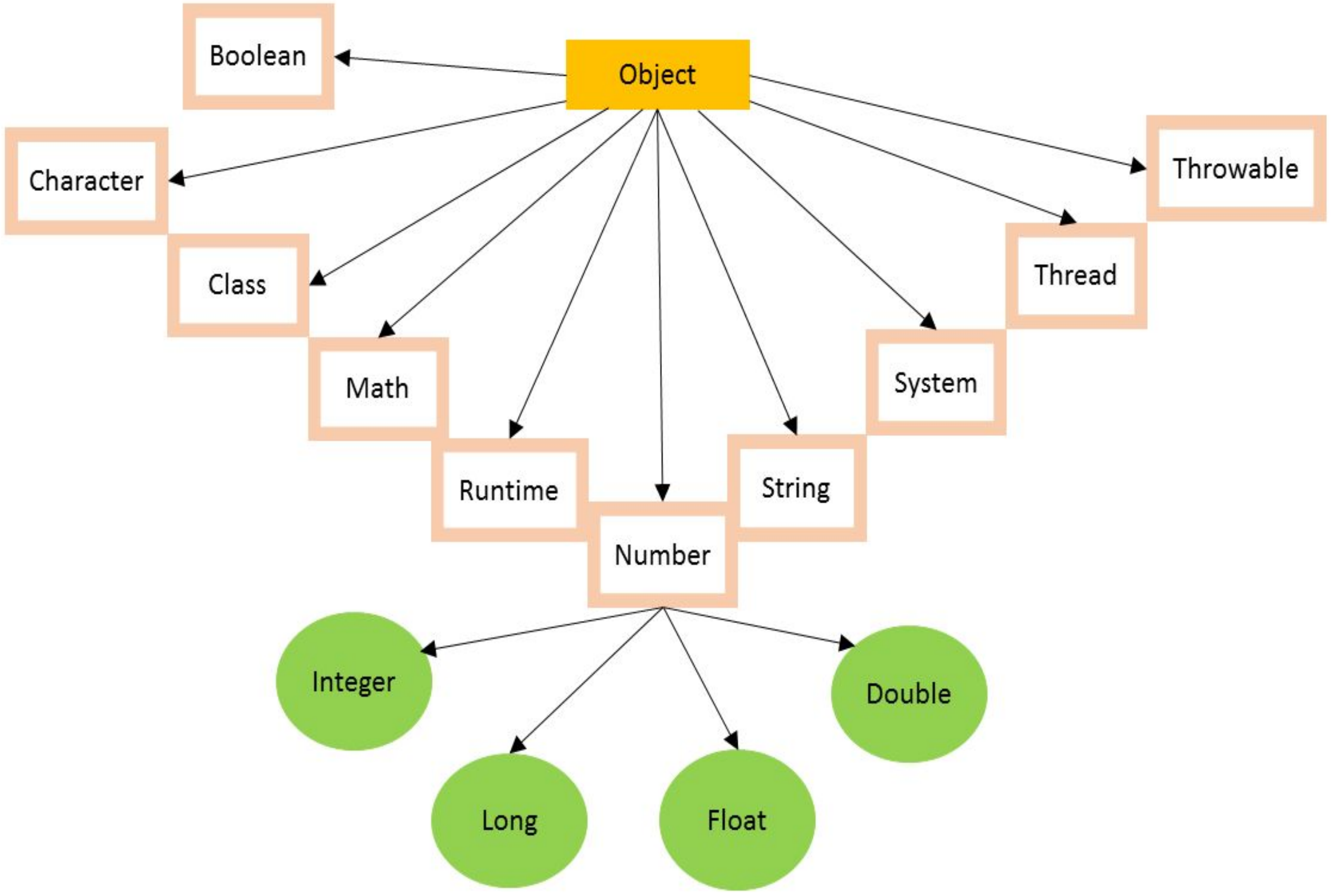
```
class Shape
{
    void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape
{
    void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape
{
    void draw(){System.out.println("drawing circle...");}
}
```

```
class Triangle extends Shape
{
    void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2
{
public static void main(String args[])
{
    Shape s;  // super class object
    s=new Rectangle();
    s.draw();
    s=new Circle();
    s.draw();
    s=new Triangle();
    s.draw();
}
}
```

Object class

- The **Object class is the parent class of all the classes** in java by default.
- **Object** class is present in **java.lang** package.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived.

Object class



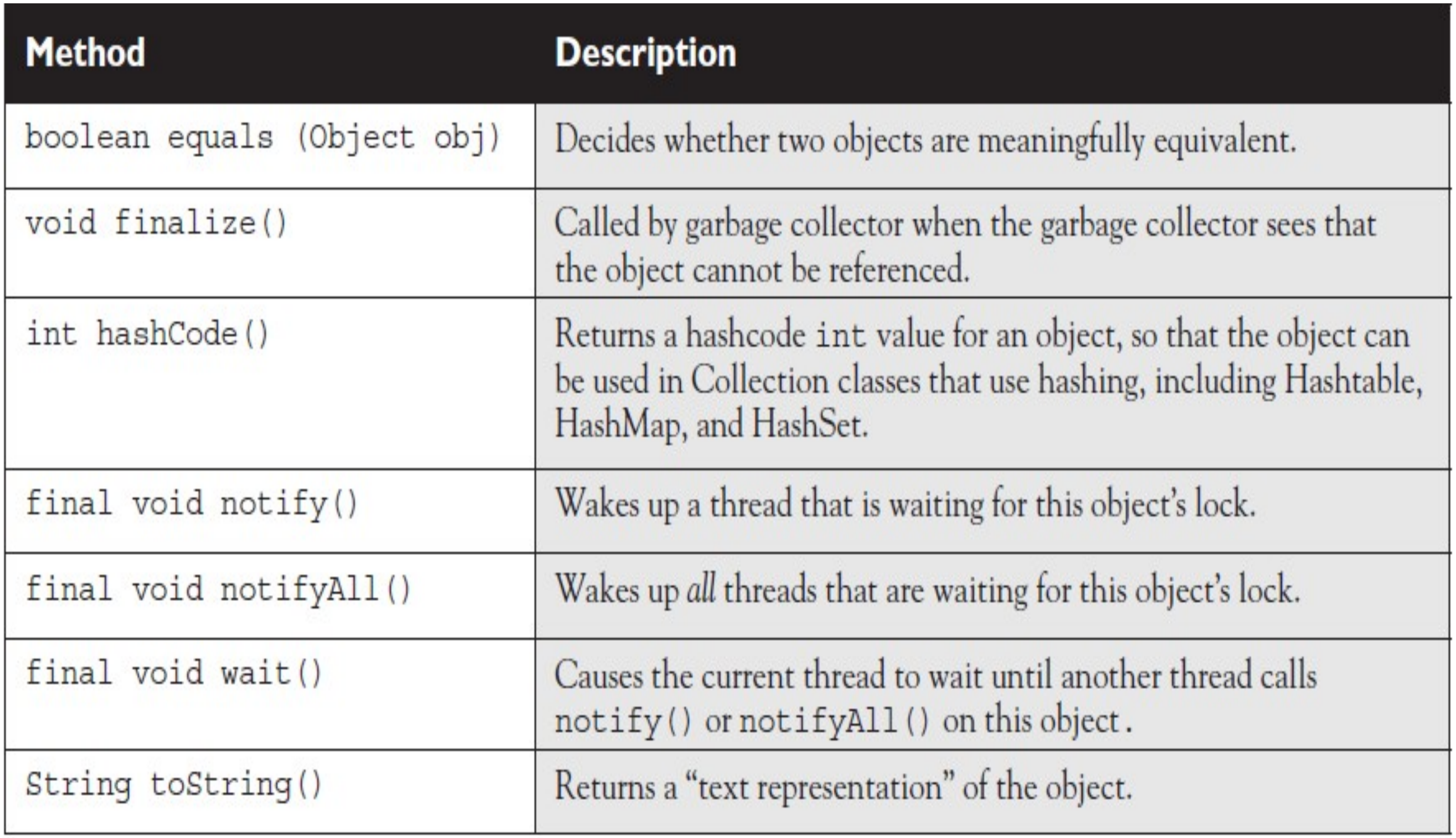
Object class

- The **object class** has several methods, like get current object, object cloning, object notified etc.
- **Example:**

```
Object obj = getObject();
```

- This method **return object of particular class type** like students, employees etc.
- Some of that methods are given below.

Object class



Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by garbage collector when the garbage collector sees that the object cannot be referenced.
`int hashCode()`	Returns a hashcode `int` value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet.
`final void notify()`	Wakes up a thread that is waiting for this object's lock.
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock.
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this object.
`String toString()`	Returns a "text representation" of the object.