Objectives

- In this session, you will learn to:
  - Use virtual method invocation
  - Explore polymorphism
  - Use the instanceof operator
  - Use casting
  - Override methods of the Object class
  - Enable generalization

Virtual Method Invocation

- An object's methods are associated to it either at compile time or at runtime.
- Its behavior is determined by its runtime reference.
- This is known as virtual method invocation.
- The following code snippet shows the implementation of virtual method invocation:

Here, the superclass object, `e`, holds the reference to the instance of the subclass.

```
Employee e = new Manager (102, "Joan Kern",
"012-23-4567", 110_450.54, "Marketing");
System.out.println (e.getDetails());
```

Method invoked virtually

The object's type is determined to be the `Manager` type at runtime. Thus, the `getDetails()` method of the `Manager` class is called.

Applying Polymorphism

- Polymorphism:
  - Ability to create a variable, a function, or an object that has more than one form
- The following embedded Word document shows a class that calculates stock grants for employees based on their role.
- If the number of employee roles increase, new methods need to be added.
- The programming approach used in the preceding scenario is not object-oriented.
- To resolve the preceding problem, write methods that accept generic parameters.

Applying Polymorphism (Contd.)

- The following code snippet refines the `EmployeeStockPlan` class by using polymorphism:

```
public class
EmployeeStockPlan
    {
    public int grantStock (Employee e)
      {
      // perform a calculation based
      on
          //Employee data
            }
    }
```

The `Employee` is superclass of all employee roles.

Using the instanceof Keyword

- The `instanceof` operator determines an object’s type at runtime, as shown in the following code snippet:

```
public class EmployeeRequisition {
public boolean canHireEmployee(Employee e)
  {            if(e instanceof Manager)
  {
          return true;
  } else {
   return false;
        }
     }
  }
```

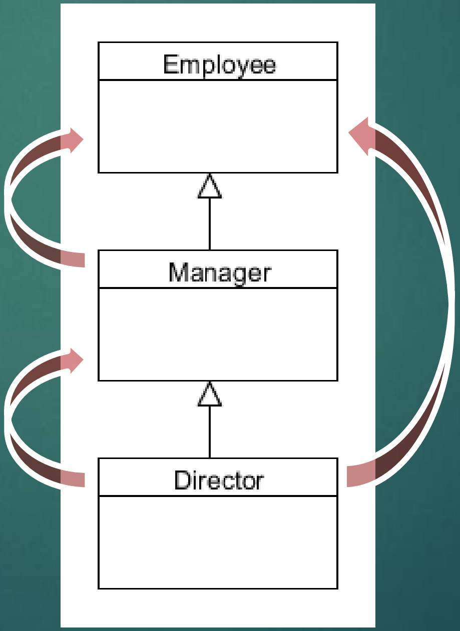
Casting Object References

- Superclass:
  - Object can hold the reference of its subclass
  - Object reference can call a method of the subclass that does not exist in it
  - Object reference must be cast to its subclass type to call the subclass method, as shown in the following code snippet:

```
public void
modifyDeptForManager (Employee e,
  String dept) {
 if (e instanceof Manager) {
 Manager m = (Manager) e;
 m.setDeptName(dept);
 }
 }
```

Casting Rules

- Casting can be of the following types:
  - Upward
  - Downward
- The following figure depicts upward casting of the object.



```
Employee e = m; // OK
```

```
Manager m = d; // OK
```

```
Director d=new Director();
Manager m=new Manager();
```

Here, `d` is an object of the `Director` class and `m` is an object of the `Manager` class.

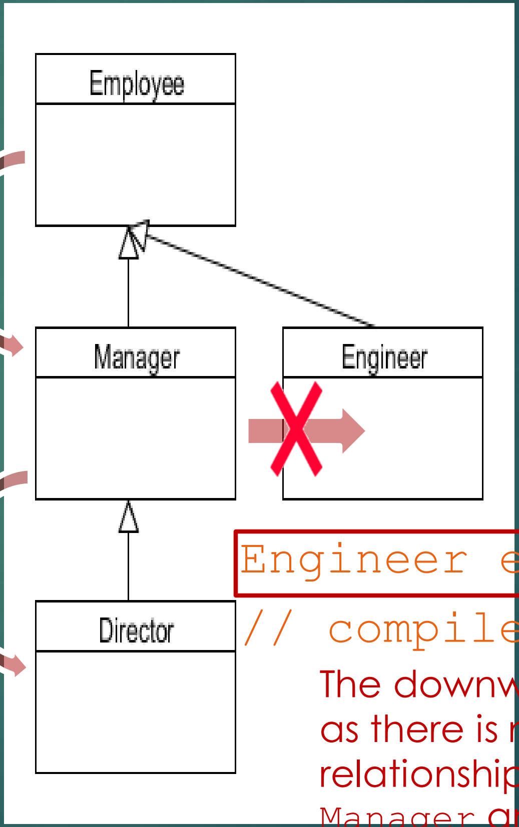
```
Employee e = d;//Ok
```

Casting Rules (Contd.)

- The following figure depicts downward casting of an object.

Casting operator

```
Manager m = (Manager) e;
// Would also work if
// e was a Director obj
```



```
Director d = (Director) m;
// fails at run time
```

The statement compiles, but throws a runtime exception. However, if `m` is an object of the `Director` class, the downward cast will succeed.

```
Engineer eng = (Engineer)m;
// compiler error
```

The downward cast is invalid, as there is no hierarchical relationship between the `Manager` and the `Engineer` classes.

Overriding Object Methods

- `java.lang.Object` class:
  - Parent of all Java classes by default
  - Contains several methods
  - Has the following important non-final methods that can be overridden:
    - `toString()`
    - `equals()`
    - `hashCode()`
- The following code snippets show the inheritance of the `Object` class in a user-defined `Employee` class:

```
public class Employee { //... }
```

Or

```
public class Employee extends Object { //... }
```

Object toString() Method

- `toString()` method:
  - Called to return the `string` value of an object
  - Can be overridden to provide instance information
- The following code snippet shows how to override the `toString()` method in the `Employee` class:

```
public class Employee{
//fields of Employee class
public String toString () {
     return "Employee id:  " + empId + "\n"
            "Employee name:" + name;
  }
}
```

Here, `empId` and `name` are data fields of the `Employee` class.

Object equals() Method

- The `equals()` method of the `Object` class compares only the object references.
- If `x` and `y` are two objects of a class, then `x` is equal to `y`
  if and only if `x` and `y` refer to the same object.
- To test the contents of the objects, instead of their references, override the `equals()` method.

Overriding Object hashCode()

- `hashcode()` method:
  - Must return the same `hashcode` value for the objects that are considered equal by the `equals()` method
  - Must be overridden, if the `equals()` method of the class is overridden
- The following code snippet shows how to override the `hashcode()` method:

```
 public int hashCode() {
int hash = 7;
hash = 83 * hash + this.empId;
hash = 83 * hash + Objects.hashCode(this.name);
hash = 83 * hash + Objects.hashCode(this.ssn);
hash = 83 * hash +
(int)(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
 return hash; }
```