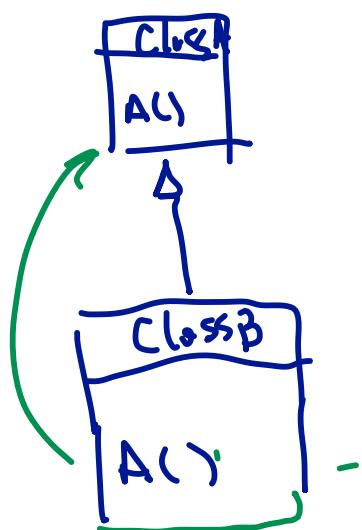


inheritance
polymorphism
encapsulation

overloading ✓ in the same class many methods
with same name but different
signature

Overriding → Cancelling the inherited method
signature is same (interface)



@Override
public void A(){}

Accessibility of Overridden Methods

An overriding method must provide at least as much access as
the overridden method in the parent class. → ✓

```
public class Employee {  
    //... other fields and methods  
    public String getDetails() { ... }  
}
```

public

```
public class Manager extends Employee {  
    //... other fields and methods  
    private String getDetails() { //... } // Compile time error
```

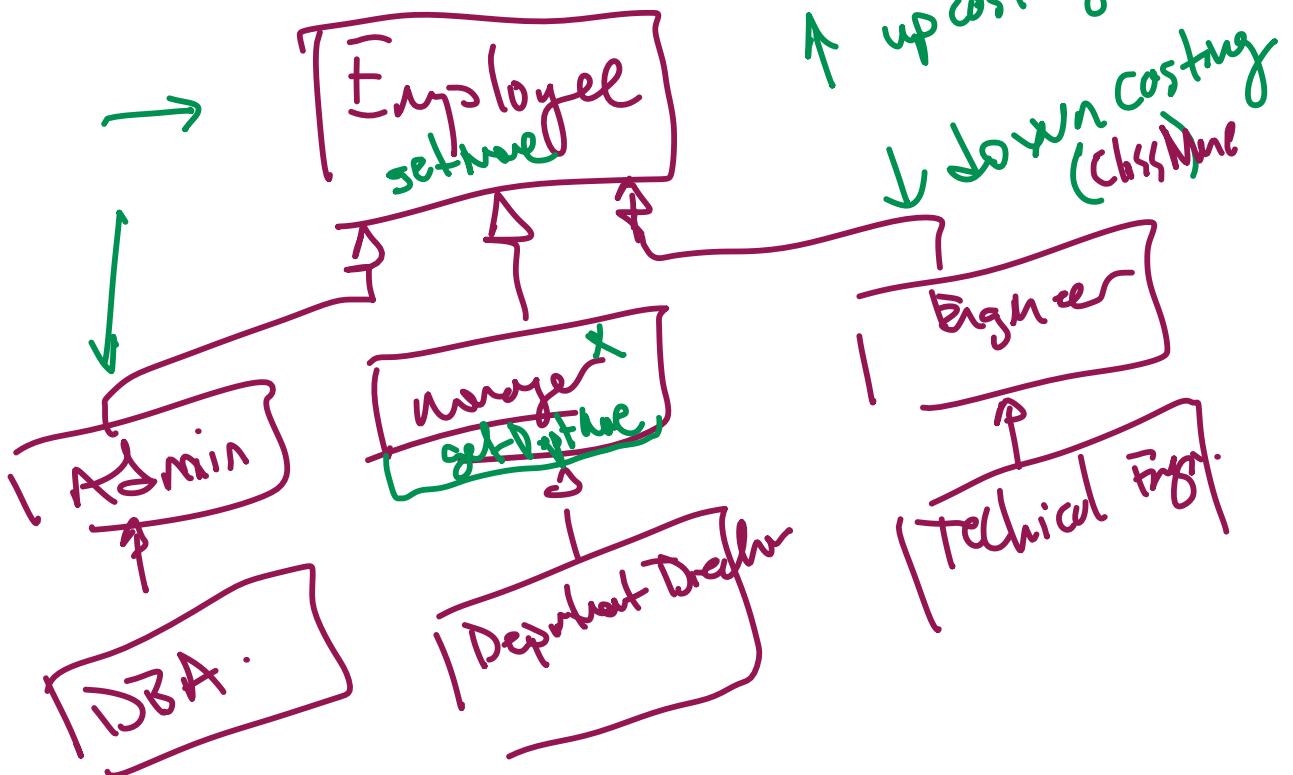
Applying Polymorphism

Suppose that you are asked to create a new class that calculates a stock grant for employees based on their salary and their role (manager, engineer, or admin): ↗

```
1 public class EmployeeStockPlan {  
2     public int grantStock (Manager m) {  
3         // perform a calculation for a Manager  
4     }  
5     public int grantStock (Engineer e) {  
6         // perform a calculation for an Engineer  
7     }  
8     public int grantStock (Admin a) {  
9         // perform a calculation for an Admin  
10    }  
11    //... one method per employee type  
12}
```

future maintenance ↗

✓
Dont
Repeat
Yourself



Applying Polymorphism

A good practice is to pass parameters and write methods that use the most generic form of your object as possible.

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

↑ up casting

```
// In the application class  
EmployeeStockPlan esp = new EmployeeStockPlan ();  
Manager m = new Manager ();  
int stocksGranted = grantStock (m);  
...
```

Dep. Director d -
new GUI

Applying Polymorphism ✓

Adding a method to Employee allows EmployeeStockPlan to use polymorphism to calculate stock.

```
public class Employee {  
    protected int calculateStock() { return 10; }  
}
```

```
public class Manager extends Employee {  
    public int calculateStock() { return 20; }  
}
```

```
public class EmployeeStockPlan {  
    private float stockMultiplier; // Calculated elsewhere  
    public int grantStock (Employee e) {  
        return (int) (stockMultiplier * e.calculateStock());  
    }  
}
```

4 - 14

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

ORACLE

public class DBA {
 protected int calculateStock() {
 return 7; }
}
DBA d = new DBA();
ESP. grantStock(d);

Using the instanceof Keyword

The Java language provides the instanceof keyword to determine an object's class type at run time.

```
1 public class EmployeeRequisition {  
2     public boolean canHireEmployee (Employee e) {  
3         if (e instanceof Manager) {  
4             return true;  
5         } else {  
6             return false;  
7         }  
8     }  
9 }
```

instanceof

Casting Object References

In order to access a method in a subclass passed as an generic argument, you must cast the reference to the class that will satisfy the compiler:

```
1 public void modifyDeptForManager (Employee e, String dept) {  
2     if (e instanceof Manager) {  
3         Manager m = (Manager) e;  
4         m.setDeptName (dept);  
5     }  
6 }
```

Cast

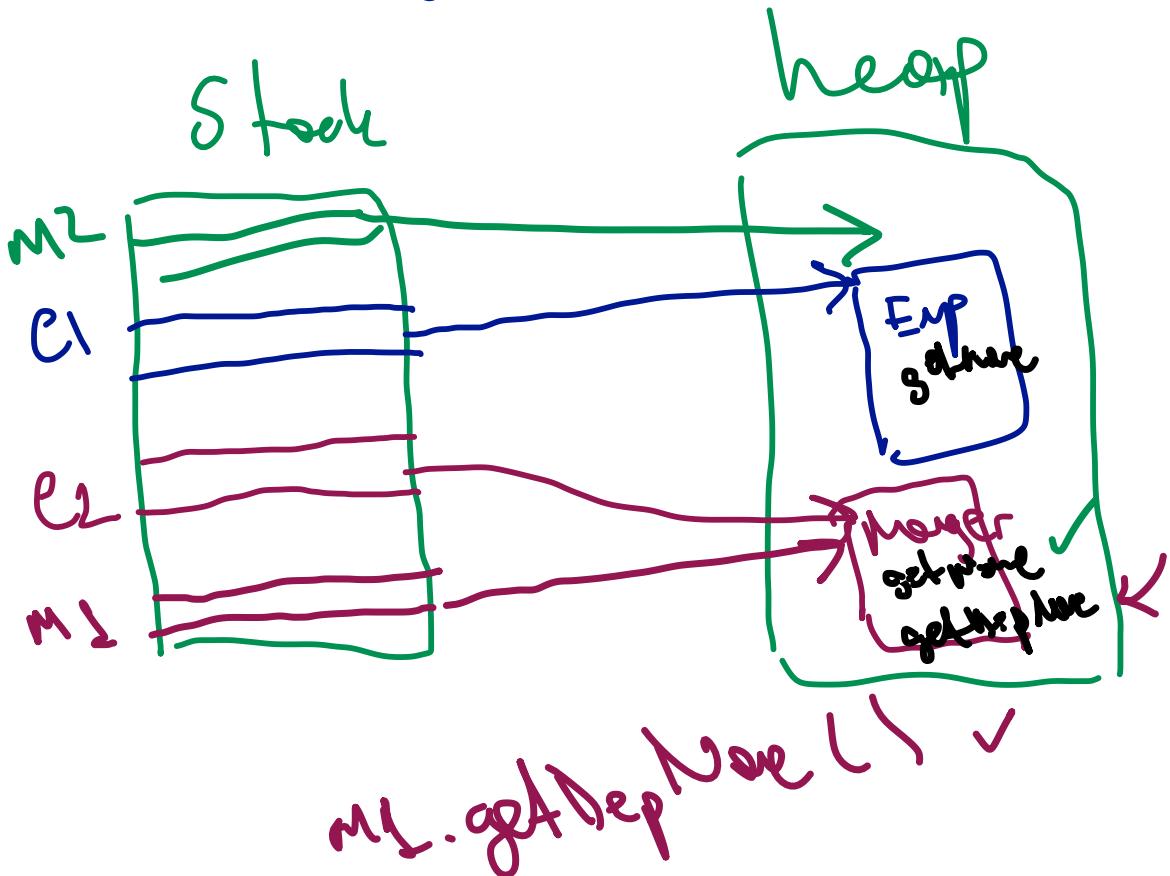
Without the cast to Manager, the setDeptName method would not compile.

$\checkmark \times$ Employee $e1 = \underline{\text{new Employee}}();$ ✓
 \checkmark Employee $e2 = \text{new Manager}();$

~~Manager~~
~~e2~~ ✓

~~Manager~~ $m1 = (\text{Manager}) e1;$ ✗
~~Manager~~ $m2 = (\text{manager}) e1;$ ✗

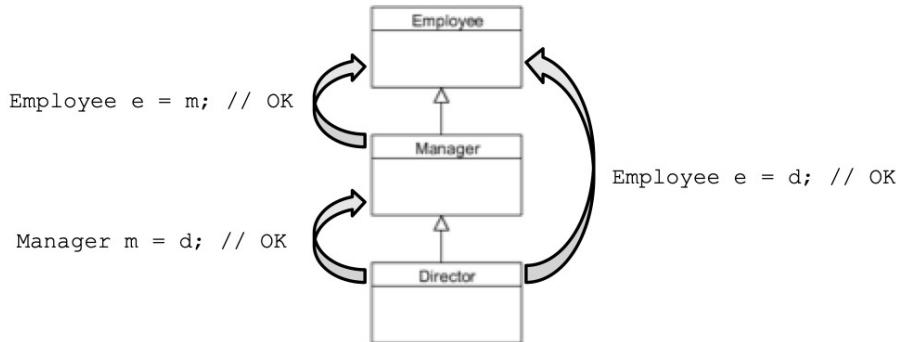
⚠



Casting Rules

Upward casts are always permitted and do not require a cast operator.

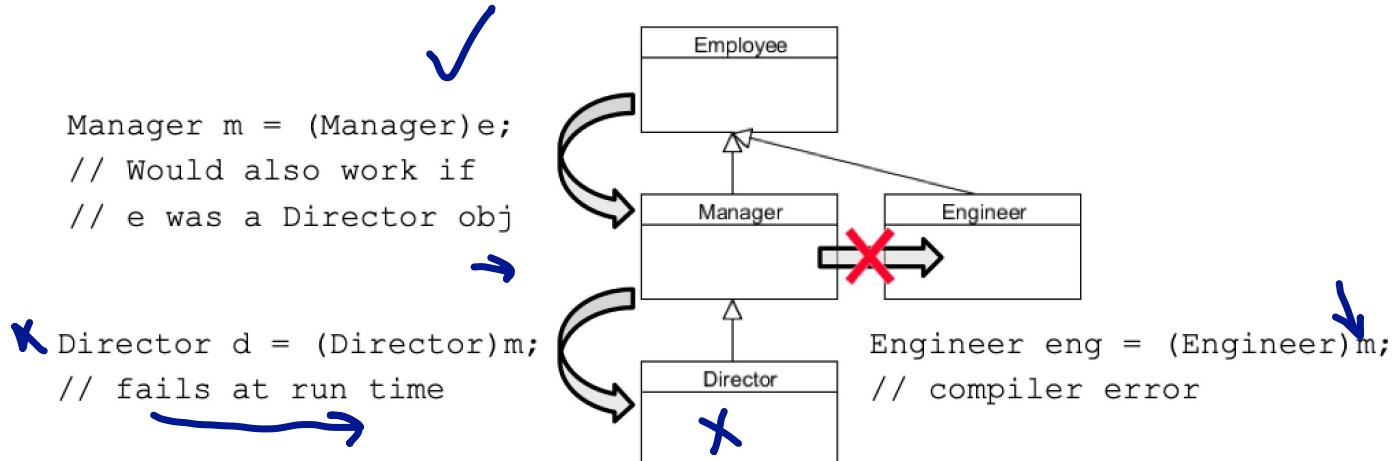
```
Director d = new Director();
Manager m = new Manager();
```



Casting Rules

For downward casts, the compiler must be satisfied that the cast is at least possible.

```
Employee e = new Manager(); ✓
Manager m = new Manager(); ✓
```



Overriding Object methods



One of the advantages of single inheritance is that every class has a parent object by default. The root class of every Java class is java.lang.Object.

- You do not have to declare that your class extends Object. The compiler does that for you.

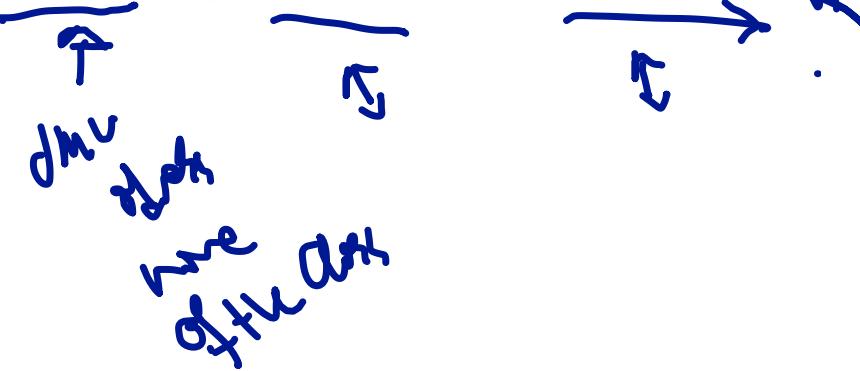
```
public class Employee { //... }
```

is equivalent to:

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

- The root class contains several non-final methods, but there are three that are important to consider overriding:

- toString, equals, and hashCode



Object toString Method



The toString method is called to return the string value of an object. For example, using the method `println`:

```
Employee e = new Employee (101, "Jim Kern", ...)  
System.out.println (e);
```

e.toString()

- String concatenation operations also invoke toString:

```
String s = "Employee: " + e;
```

- You can use toString to provide instance information:

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

overriding

- This is a better approach to getting details about your class than creating your own `getDetails` method.

Object equals Method

$m == m$

The Object equals method compares only object references.

- If there are two objects x and y in any class, x is equal to y if and only if x and y refer to the same object.
- Example:

```
Employee x = new Employee (1, "Sue", "111-11-1111", 10.0);  
Employee y = x;  
x.equals (y); // true  
Employee z = new Employee (1, "Sue", "111-11-1111", 10.0);  
x.equals (z); // false!
```

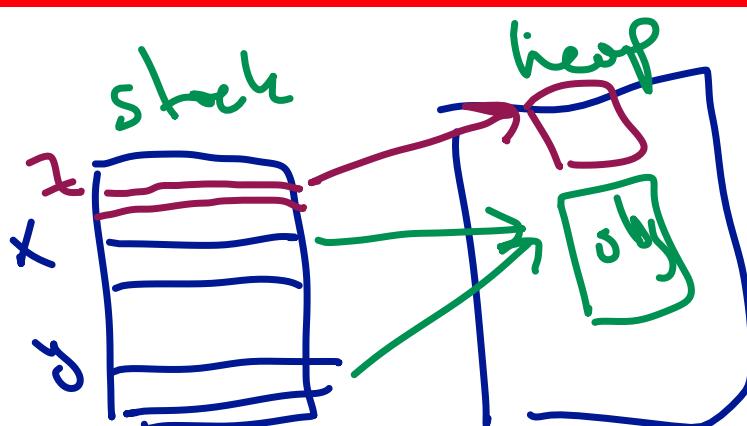
$m.equals(m)$

- Because what we really want is to test the contents of the Employee object, we need to override the equals method:

```
public boolean equals (Object o) { ... }
```



ORACLE



$x.equals(y)$
true

Overriding equals in Employee

An example of overriding the equals method in the Employee class compares every field for equality:

```
1 public boolean equals (Object o) {  
2     boolean result = false; ↴ true  
3     if ((o != null) && (o instanceof Employee)) {  
4         Employee e = (Employee)o; ↴ casting  
5         if ((e.empId == this.empId) &&  
6             (e.name.equals(this.name)) &&  
7             (e.ssn.equals(this.ssn)) &&  
8             (e.salary == this.salary)) {  
9                 result = true;  
10            }  
11        }  
12        return result;  
13    }
```

Overriding Object hashCode

The general contract for Object states that if two objects are considered equal (using the equals method), then integer hashCode returned for the two objects should also be equal.

```
1 // Generated by NetBeans  
2 public int hashCode() {  
3     int hash = 7; ✓  
4     hash = 83 * hash + this.empId;  
5     hash = 83 * hash + Objects.hashCode(this.name);  
6     hash = 83 * hash + Objects.hashCode(this.ssn);  
7     hash = 83 * hash +  
8             (int) (Double.doubleToLongBits(this.salary) ^  
9             (Double.doubleToLongBits(this.salary) >>> 32));  
10    return hash;  
11 }
```

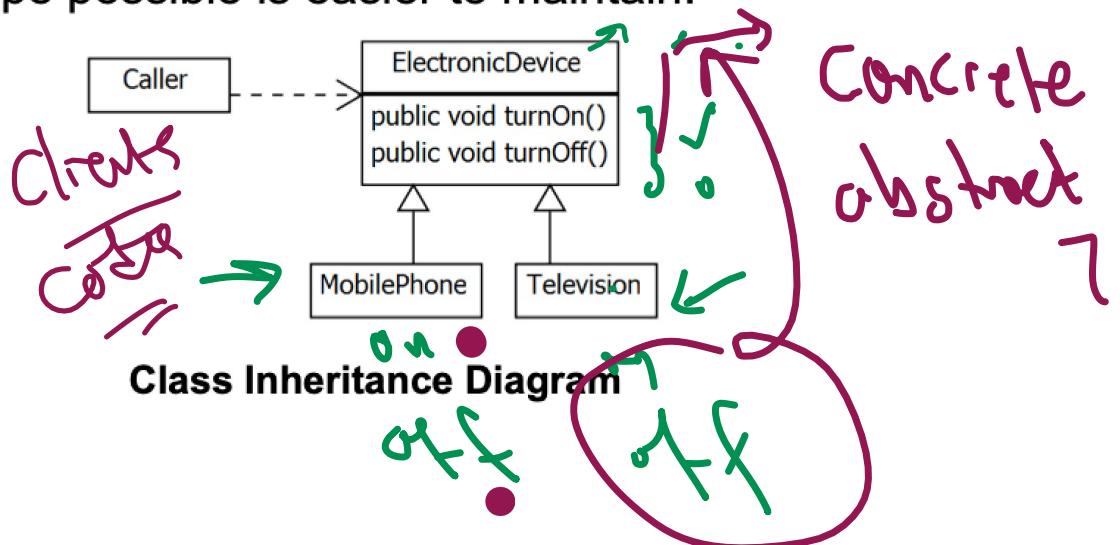
Advanced Class Design

5

Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- Method inheritance: Subclasses avoid code duplication by inheriting method implementations.
- Generalization: Code that is designed to rely on the most generic type possible is easier to maintain.



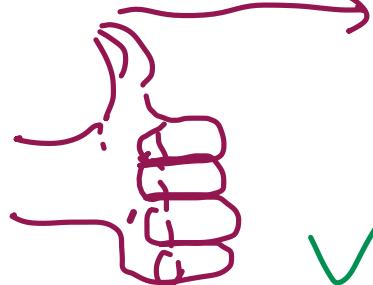
Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();  
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.

Rule of thumb



field declarations, arguments,

return types

object



+ Design Principles

→ loosely coupled ⇒

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized. → *abstract*

```
public class Television extends ElectronicDevice {  
  
    public void turnOn() {  
        changeChannel(1);  
        initializeScreen();  
    }  
    public void turnOff() {}  
  
    public void changeChannel(int channel) {}  
    public void initializeScreen() {}  
  
}
```

Defining Abstract Classes

interface

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class cannot be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```



polymorphism, template ~~method~~, class

both codes, and abstract method

Defining Abstract Methods

A method can be declared as abstract by using the **abstract** method-level modifier.

```
public abstract class ElectronicDevice {  
    public abstract void turnOn();  
    public abstract void turnOff();  
}
```

No braces

An abstract method:

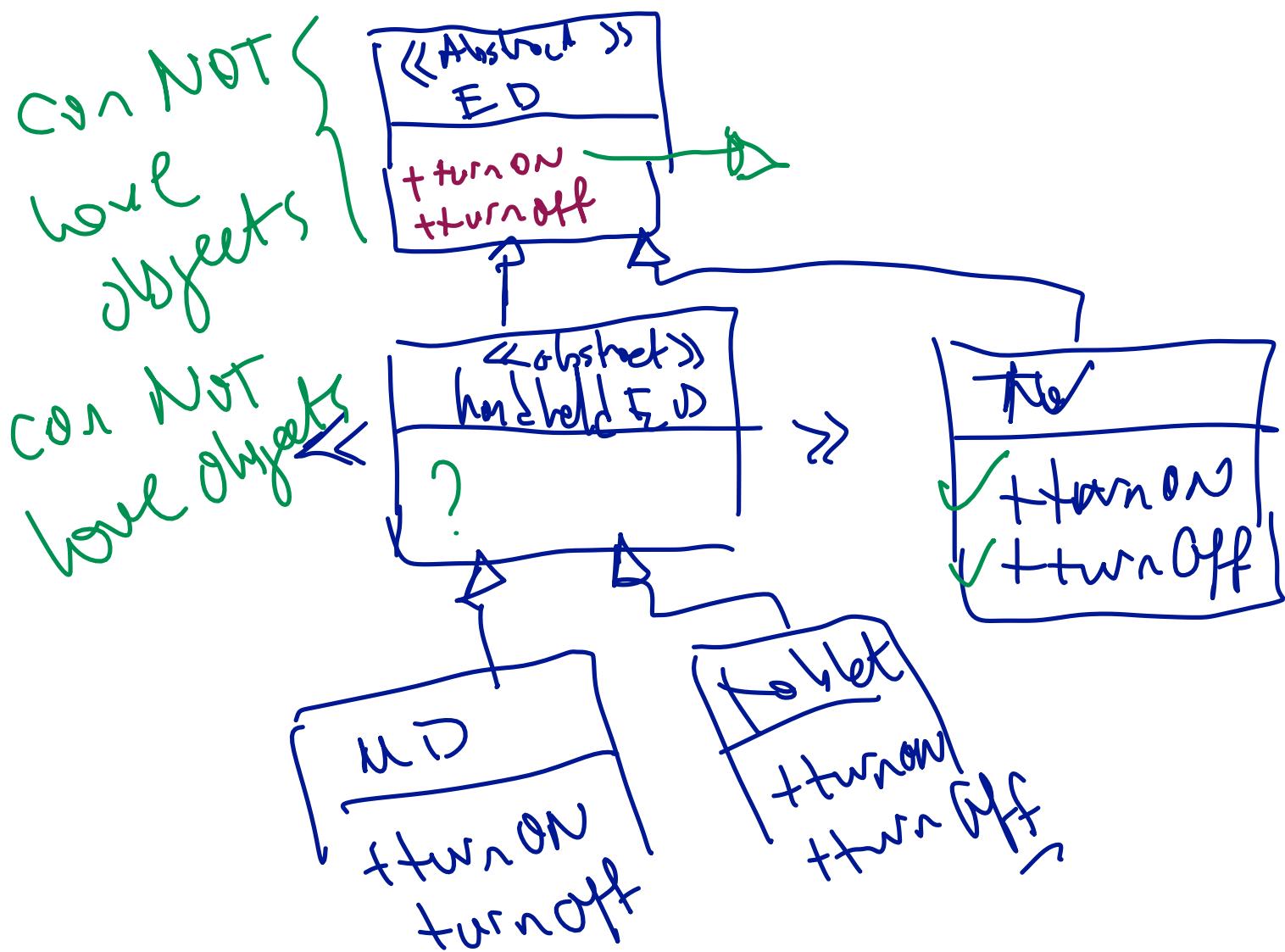
- Cannot have a method body →
 - Must be declared in an abstract class →
 - Is overridden in subclasses →
-

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and non-abstract methods. ✓
- When inheriting from an abstract class, you must do either of the following: →
 - Declare the child class as abstract. ✓
 - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override  
abstract method turnOn() in ElectronicDevice
```



static Keyword

The static modifier is used to declare fields and methods as class-level resources. Static class members:

- Can be used without object instances ✓
- Are used when a problem is best solved without objects ✓
- Are used when objects of the same type need to share fields ↴
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason → ✓

Moth.

Java
class
static

public static void main(String[] args)
.
.
.
.

Class level resources

Object level resources

Static Methods

Belongs Class

Static methods are methods that can be called even if the class they are declared in has not been instantiated. Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented.
 - `java.lang.Math` contains many static methods
- Are commonly used in place of constructors to perform tasks related to object initialization
- Cannot access non-static members within the same class
- Can be hidden in subclasses but not overridden
 - No virtual method invocation

public class

✓ static int a = 10;

✓ int b = 20;

test { shared

among objects
instances

public static void A() {

b won't be accessible

but a will be accessible

}

}

Implementing Static Methods

```
public class StaticErrorClass {  
    private int x;  
  
    public static void staticMethod() {  
        x = 1; // compile error  
        instanceMethod(); // compile error  
    }  
  
    public void instanceMethod() {  
        x = 2;  
    }  
}
```

Annotations:

- Green curly braces group the entire class definition.
- Green circles highlight the keyword `static` in `staticMethod()`.
- Red boxes highlight the errors: `x = 1; // compile error` and `instanceMethod(); // compile error`.
- A green curly brace groups the `staticMethod()` block, labeled "Class Method".
- A green curly brace groups the `instanceMethod()` block, labeled "Object Method".

Calling Static Methods

```
(6-1)  
double d = Math.random();  
StaticUtilityClass.printMessage();  
StaticUtilityClass uc = new StaticUtilityClass();  
uc.printMessage(); // works but misleading  
sameClassMethod();
```

Annotations:

- Pink arrows point to the first three lines of code.
- A pink arrow points to the `uc.printMessage();` line, with the note: "works but misleading".
- A pink arrow points to the `sameClassMethod();` line.

When calling static methods, you should:

- Qualify the location of the method with a class name if the method is located in a different class than the caller
 - Not required for methods within the same class
- Avoid using an object reference to call a static method

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated. Static variables are: —

- Called class variables
- Limited to a single copy per JVM
- Useful for containing shared data
 - Static methods store data in static variables.
 - All object instances share a single copy of any static variables.
- Initialized when the containing class is first loaded

Before executing $\mathbf{m1} \dots - - -$
methods, all class with
static fields and methods
are loaded to the JVM
 $\Leftarrow \underline{\text{Math.sqrt}(s)}$

Defining Static Variables

2 minutes

```
public class StaticCounter {  
    private static int counter = 0;  
  
    public StaticCounter() {  
        counter++;  
    }  
  
    public static int getCount() {  
        return counter;  
    }  
}
```

Only one copy in memory

Using Static Variables

```
double p = Math.PI; {3.1415...}  
  
new StaticCounter();  
new StaticCounter();  
System.out.println("count: " + StaticCounter.getCount());
```

2 ✓

When accessing static variables, you should:

- Qualify the location of the variable with a class name if the variable is located in a different class than the caller
 - Not required for variables within the same class
- Avoid using an object reference to access a static variable

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;  
import static java.lang.Math.*;
```

- Calling the Math.random() method can be written as:

```
public class StaticImport {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}
```

↑

Math .

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {  
    public final void printMessage() {  
        System.out.println("This is a final method");  
    }  
}
```

```
public class MethodChildClass extends MethodParentClass {  
    // compile-time error  
    public void printMessage() {  
        System.out.println("Cannot override method");  
    }  
}
```

final in front of

- + class = can not be subclasses
- + method = can not be overridden
- + field = constant

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error  
public class ChildClass extends FinalParentClass { }
```

Final Variables

The `final` modifier can be applied to variables. Final variables may not change their values after they are initialized. Final variables can be:—

- Class fields
 - Final fields with compile-time constant expressions are constant variables.
 - Static can be combined with final to create an always-available, never-changing variable.
- Method parameters →
- Local variables →

Note: Final references must always reference the same object, but the contents of that object may be modified.



Declaring Final Variables



```
public class VariableExampleClass {  
    private final int field; 100  
    private final int forgottenField;  
    private final Date date = new Date();  
    public static final int JAVA_CONSTANT = 10;  
  
    public VariableExampleClass() {  
        field = 100;  
        // compile-time error - forgottenField  
        // not initialized  
    }  
  
    public void changeValues(final int param) {  
        param = 1; // compile-time error  
        date.setTime(0); // allowed ✓  
        date = new Date(); // compile-time error ✓  
        final int localVar;  
        localVar = 42; ✓  
        localVar = 43; // compile-time error  
    }  
}
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.



When to Avoid Constants

public static final variables can be very useful, but there is a particular usage pattern you should avoid. Constants may provide a false sense of input validation or value range checking.

- Consider a method that should receive only one of three possible values:

```
Computer comp = new Computer();  
comp.setState(Computer.POWER_SUSPEND);
```

This is an int constant
that equals 2.

- The following lines of code still compile:

```
Computer comp = new Computer();  
comp.setState(42);
```

static field
final

↑
→
enum writers instead of final static

Typesafe Enumerations

Java 5 added a typesafe enum to the language. Enums:

- Are created using a variation of a Java class
- Provide a compile time range check

```
public enum PowerState {  
    OFF,  
    ON,  
    SUSPEND;  
}
```

These are references to the only three PowerState objects that can exist.

An enum can be used in the following way:

```
Computer comp = new Computer();  
comp.setState(PowerState.SUSPEND);
```

This method takes a PowerState reference .

Enum Usage

Enum references can be statically imported.

```
import static com.example.PowerState.*;  
  
public class Computer extends ElectronicDevice {  
    private PowerState powerState = PowerState.OFF;  
    //...  
}
```

PowerState.OFF

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {  
    switch(state) {  
        case OFF:  
        //...  
    }  
}
```

or

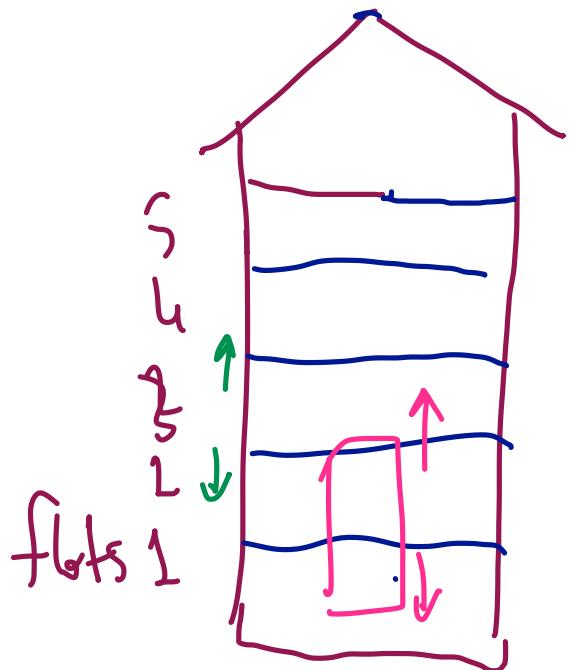
Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {  
    OFF("The power is off"),  
    ON("The usage power is high"),  
    SUSPEND("The power usage is low");  
  
    private String description;  
    private PowerState(String d) {  
        description = d; →  
    }  
    public String getDescription() {  
        return description;  
    }  
}
```

Call a PowerState constructor to initialize the public static final OFF reference.

The constructor may not be public or protected.



- + Simulate an elevator has capacity of number
- + Apartment Max flat number
- + min flat number

- ⊕ Apartment has elevators(s)
- ⊕ Elevator When it is moving It becomes busy
- ⊕ Elevator keep track of the flat they are in.
- ⊕ Elevator can move up or down
- ⊕ Elevator keeps track of # of passenger



- * Every ninja has an attack point starts 5
- * Every ninja has an health point starts 10
- * Every ninja can attack %50
- * Every ninja can defend %50
(block the successful attack)
- * Each successfull attack if not
not blocked will decrease
health point of defeder as
attack points of attacker

Ⓐ Ninja can either attack or defend at a time.

Ⓑ → They are attacking and defending in order

→ Roll a dice with 100 faces

for each ninja

bigger value of dice will attack the other will defend
if they have same dice rolled they roll again.