Inheritance
Part 2

LASSONDE SCHOOL OF ENGINEERING | YORK UNIVERSITÉ UNIVERSITY | EECS ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Object Class

- The Object class is the root of all inheritance hierarchies
- The Object class defines methods applicable to and required by all Java classes.
- To ensure all classes have these obligatory methods, all classes implicitly extend the Object class

Obligatory methods

- toString
  - Returns a String representation of the object
  - Object implementation outputs memory address
- equals
  - Determines the equality between two objects
  - Object implementation compares memory address
- hashCode
  - Provides a pseudo-unique code
  - Determines location in hashmap or hashset

- *Additional methods not listed here (see API)*

Collections and Inheritance

- Specifying a parameterized type for a collection will allow elements of that type (or derived from that type)

```
ArrayList<BankAccountV4> list =
                        new ArrayList<BankAccountV4();
list.add(new BankAccountV4(…
list.add(new RewardAccount(…
list.add(new BankAccountV4(…
list.add(new RewardAccount(…

for (BankAccountV4 ba : list)
{
      System.out.println(ba.getBalance());
}
```

The Substitutability Principle

- "When a parent is expected, a child is accepted"

- This allows the same code to process both parent classes and their descendants
- For example, a program intended to handle BankAccountV4 objects will be able to handle RewardAccount objects without modification

Substitutability Example

- The following is correct:
  - `BankAccountV4 ba1 = new BankAccountV4(…`
  - `BankAccountV4 ba2 = new RewardAccount(…`

  - Subsequently, any method that can be called on a `BankAccountV4` can also be called on a `RewardAccount`

- The following is NOT correct (why?):
  - `RewardAccount ra = new BankAccountV4(…`

Determining Type

- To determine the data type of an object, use
  - `instanceof` operator
  - a combination of `getClass()` and `.class`

- Examples on subsequent slides

Determining Type (2)

- Using `instanceof`:

```
BankAccountV4 ba1 = new BankAccountV4(…
BankAccountV4 ba2 = new RewardAccount(…
```

`ba1 instanceof BankAccountV4` → true

`ba2 instanceof RewardAccount` → true

`ba2 instanceof BankAccountV4` → true
(by substitutability)

`ba1 instanceof RewardAccount` → false

Determining Type (3)

- Using `getClass()` and `.class`:

```
BankAccountV4 ba1 = new BankAccountV4(…
BankAccountV4 ba2 = new RewardAccount(…

ba1.getClass() == ba1.getClass() → true
ba1.getClass() == ba2.getClass() → false

ba1.getClass() == BankAccountV4.class → true
ba2.getClass() == BankAccountV4.class → false
ba2.getClass() == RewardAccount.class → true
```

The Need to Cast

- Wrong:
  - `BankAccountV4 ba = new RewardAccount(…`
    `balance = ba.getPointBalance()`
  - Early binding will fail because `BanckAccountV4` does not have a `getPointBalance()` method
- Correct:
  - `BankAccountV4 ba = new RewardAccount(…`

```
if (ba instanceof RewardCard)
{
   balance = ((RewardCard) ba).getPointBalance();
}
```

Early and Late Binding

- Binding: validation of a method call

- Early binding:
  - Occurs at compile-time
  - Binding failure results in a compile-time error (i.e., cannot find method)

- Late binding:
  - Applicable only when (explicit) inheritance is used
  - Occurs at run-time

Binding Example One

```
BankAccountV4 ba = new RewardAccount(…
ba.getBalance();
```

- Early binding:
  - Verifies `getBalance()` method in `BankAccountV4` class
- Late binding:
  - Determines `ba` points to a `RewardAccount` object
  - Cannot find `getBalance()` method in `RewardAccount` because `getBalance()` was not overridden in `RewardAccount`
  - Calls `getBalance()` method in `BankAccountV4` class

Binding Example Two

```
BankAccountV4 ba = new RewardAccount(…
ba.deposit();
```

- Early binding:
  - Verifies `deposit()` method in `BankAccountV4` class
- Late binding:
  - Determines `ba` points to a `RewardAccount` object
  - Calls `deposit()` method in `RewardAccount` class instead

Polymorphism

- The ability of a method to take on various forms

- Occurs when early binding targets a method in a parent class and late binding targets the method with the same signature in a descendant class
  - E.g., the `deposit(double amount)` method from the previous example

Abstract Classes and Interfaces

- Interfaces:
  - Define only method signatures
  - Methods have no implemented body
  - Allow implementer to define class requirements to other implementers
- Abstract classes:
  - Only some (not all) methods are implemented
  - Allow implementers implement some methods and define requirements for others
- Classes:
  - All methods are implemented

Abstract Classes and Interfaces (2)

- Classes: `public class` *`ClassName`*
- Abstract: `public abstract class` *`ClassName`*
- Interface: `public interface` *`InterfaceName`*

- Both abstract classes and interfaces can be used as types for declarations, but neither can be instantiated
  - Look for a class that extends it or a (static) method that returns a pre-made instance of it
  - E.g., Try to create an instance of Calendar

Preventing Inheritance

- Inheritance has benefits, but sometimes class features should not be changed

- E.g., Object.getClass()
  - A class should not be able to change or falsify its identity

- E.g., methods in the Math class
  - Should not be able to change how abs, sine, exp, etc. are calculated

Final Classes and Methods

- Final classes cannot be extended (no children)

```
public final class Math
{
    …
```

- Final methods cannot be overridden

```
public final Class getClass()
{
    …
```