Polymorphism

A deeper look into Java's programming model

Robert Toscano

Polymorphism

- Ability of objects belonging to different types to respond to methods of the same name
- Ability to override functionality from extended super class
- Java handles which overridden versions of methods are to be executed
- Lets have a look at some examples

The Object Class

- Every root class, that is a class that does not extend another class, implicitly extends the java.lang.Object class
- java.lang.Object contains methods that all classes inherit
- These include:
  - clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, and wait

Overriding Methods

Different than Method Overloading

- Superclass
  - If class A extends class B, then class B is the superclass of A
  - Consequently, class A is a subclass of class B
- If class B contains a method with the signature:
  - public void foo (int arg)
- Then class A can override the method by providing a method with the same signature

The equals method

- public boolean equals (Object o);
- All classes inherit this method from the Object class
- Performs reference equality (checks whether two references refer to the same object in memory)
- You must override this method if your class needs to have an idea of equality among instances

Using Object.equals method

- Two CheckingAccounts are equal if they have the same account balance
- CheckingAccount c1 = new CheckingAccount(100); CheckingAccount c2 = new CheckingAccount(100);
- c1.equals(c1); //== true
- c2.equals(c2); //== true
- c1.equals(c2); //== false

Our Own equals Method

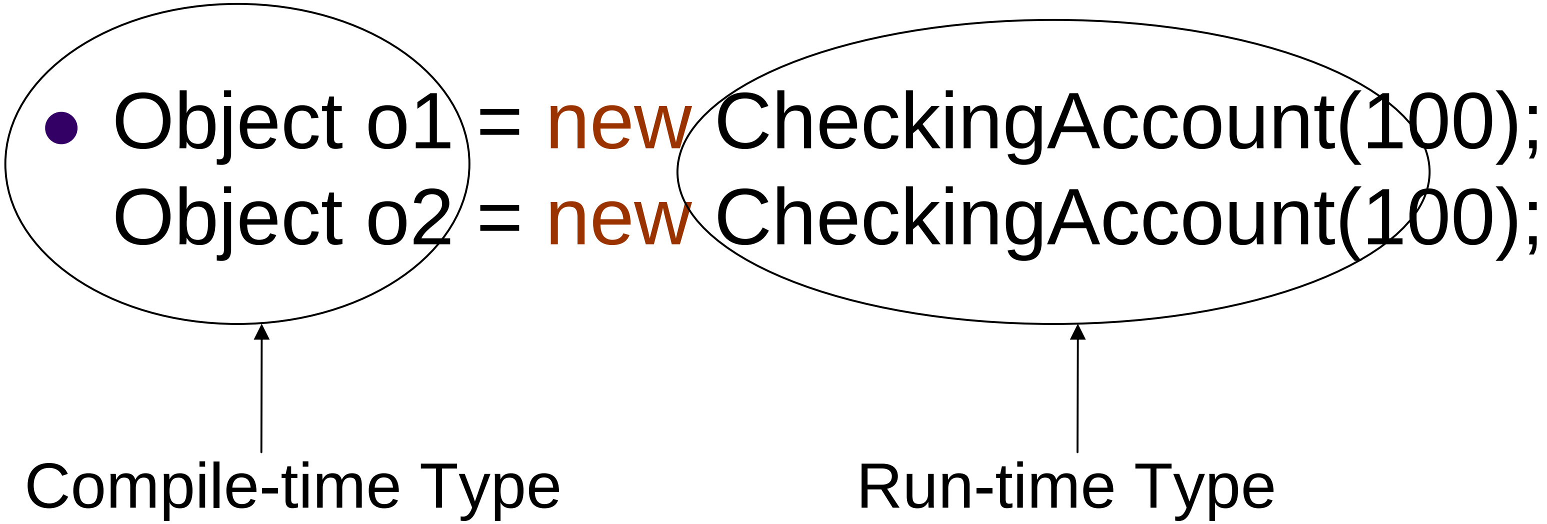
```
public class CheckingAccount extends BankAccount {
        …
        public boolean equals (Object o) {
            if (o instanceof CheckingAccount) {
                    CheckingAccount c = (CheckingAccount)o;
                    return balance == c.balance;

            } else {
                    return false;
            }
        }
        …
}
```

Using your equals method

- CheckingAccount c1 = new CheckingAccount(100); CheckingAccount c2 = new CheckingAccount(100);

- c1.equals(c1); //== true
- c2.equals(c2); //== true
- c1.equals(c2); //== true

Something Stranger



- Object o1 = new CheckingAccount(100);
  Object o2 = new CheckingAccount(100);

Compile-time Type

Run-time Type

- o1.equals(o1); //== true
- o2.equals(o2); //== true
- o1.equals(o2); //== true

Compile-time V.S. Run-time

- Compile-time type
  - Type known ahead of time, at time of writing the code—at compile time
  - During the lifetime of the program, the compile time type never changes for a given instance
- Run-time type
  - The compiler doesn't have a way of knowing what the runtime type of an object is

Method Dispatch

- Even though our objects were of **compile-time** type Object, the equals method of the CheckingAccount class was called
- This occurs because Java chooses to call the method of the instance's **run-time** type and not the compile-time type
- Let's look at another example of method dispatch

Example: BankAccount

```
public abstract class BankAccount {
        …
        public void withdraw (int amount) {…}
        …
}
```

- Now, CheckingAccount and SavingsAccount are overriding the withdraw method

Example: BankAccount

```
BankAccount b1 = new CheckingAccount(10);
BankAccount b2 = new SavingsAccount(10);

b1.withdraw(5);
//calls CheckingAccount.withdraw(int)

b2.withdraw(5);
//calls SavingsAccount.withdraw(int)
```

Example: Function Arguments

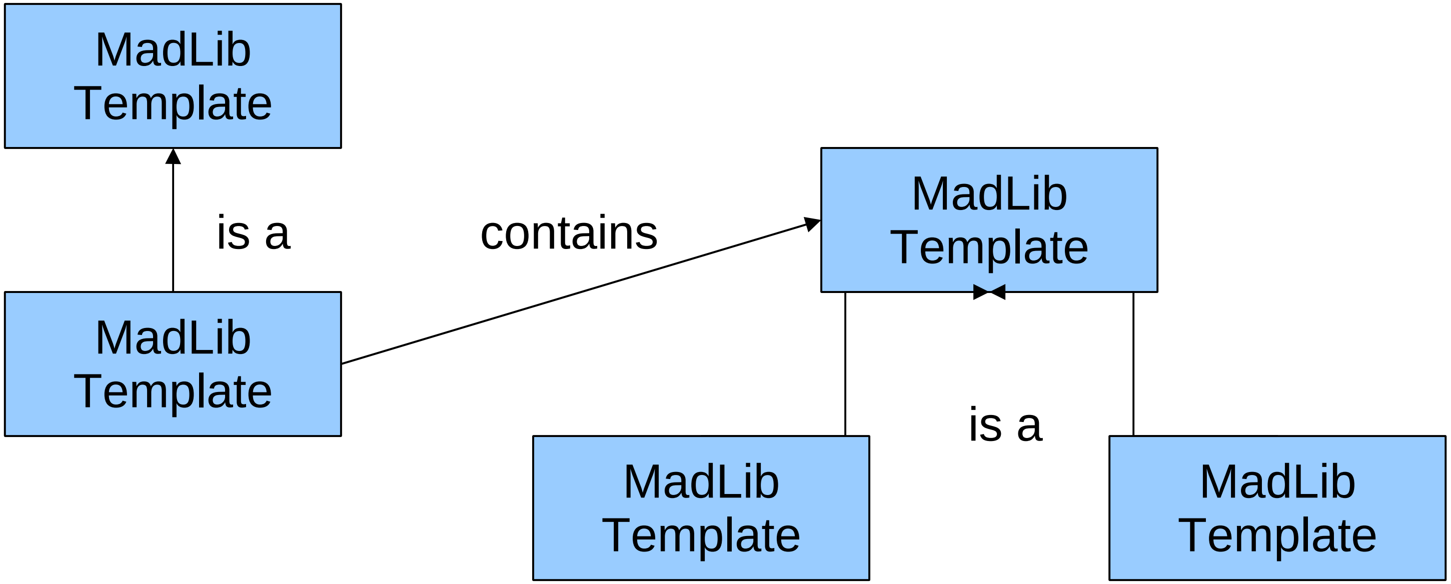
```
…
public static boolean deleteAccount (BankAccount accnt) {
            …
}
…                      cachedBrain.addTextChannel("Messages");
```

- Can pass a CheckingAccount or SavingsAccount, the compiler cannot know

Advantages Of Using General Types

- Can change the underlying implementation later
- Don’t have to change code because only use methods from more general type
- Example: Collection v.s. LinkedList and ArrayList

Mad Libs


MadLib Template
is a
contains
MadLib Template
MadLib Template
is a
MadLib Template
MadLib Template