CSSE 220

Object & Polymorphism

Import *Polymorphism* from repo

Access Levels: Review

Review On Your Own

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Access Levels: Review

Review On Your Own

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Abstract Classes:

- Are a hybrid of superclasses and interfaces
  - Like regular superclasses:
    - Provide data members
    - Provide implementation of some methods
  - Like interfaces
    - Just provide signatures and docs of other methods
    - Can't be instantiated
- Example:

```
public abstract class BankAccount {
  private int balance;
  public abstract BankAccount getCopy();
    …
}
```

Elided methods also have implementations

Abstract Classes:

UML Class Diagram Review

- Inheritance, Associations, and Dependencies

Recall UML: Associations

Register has two fields:

1. currentSale
2. Location

Names given in oval at arrowhead

Solid line, open arrowhead = "has-a"

Recall UML: Dependencies

Dependency lines are dashed

Field association lines are solid

Use association lines only when an item is stored as a field.
Solid line indicates a stronger relationship

Recall UML: Inheritance

- Generalization (superclass)
- Specialization (subclass)

Closed arrowhead = "is-a"

Two types:

- **solid line = inherits**
- **dotted line = implements**

Solid line indicates a stronger relelationship

I, Object

- The superest class in Java

Object

Every class in Java inherits from ***`Object`***

- Directly and **explicitly**:
  - `public class String extends Object {…}`
- Directly and **implicitly**:
  - `class BankAccount {…}`
- **Indirectly**:
  - `class SavingsAccount extends BankAccount {…}`

Q1

One Superclass
Possibly multiple Interfaces

Every class in Java inherits from ***`Object`***

- One 'extends' class:
  - `public class String extends Object {…}`
- Possibly multiple 'implements':
  - `class SomeClass implements I1, I2, .., IN {…}`

Q1

Object Provides Several Methods

- `String toString()`
- `boolean equals(Object otherObject)`
- `Class getClass()`
- `Object clone()`
- …

Often overridden

Sometimes useful

Correctly overriding *clone* is difficult and thought to be dangerous by many

Q2

Overriding `toString()`

- Return a String containing a concise, human-readable summary of the object's internal state

- Very useful because it's called automatically:
  - During string concatenation
  - For printing
  - In the debugger

- **`getClass().getName()` OR `getClass().getSimpleName()`** comes in handy here…

Q3

Overriding `equals(Object o)`

- `equals(Object foo)` – should return true when comparing two objects of same type with same "meaning"

- How?
  - Must check types—use **`instanceof OR getClass().isAssignableFrom(foo.getClass())`**
  - Must compare state—use **cast**

Recall casting a variable: Taking an Object of one particular type and telling the Java Runtime System to treat it as another Object type

Q4

Polymorphism

- Review and Practice

Recall: Polymorphism and Subclasses

- Exam 2 will have a question like #5 on today's quiz
- Why: You have to know the Java rules for polymorphism

```
interface Top {
    public void alpha();
    public void beta();
    public void gamma();
    public void delta();
    // Note no epsilon here
}
```

```
class One implements Top {

    public void alpha() {
        System.out.println("A");
    }

    public void beta() {
        System.out.println("B");
    }

    public void gamma() {
        System.out.println("C");
    }

    public void delta() {
        System.out.println("D");
        this.beta();
    }
}
```

```
class Two extends One
        implements Top {

    public void beta() {
        System.out.println("E");
    }

    public void gamma() {
        super.gamma();
        System.out.println("F");
    }

    public void epsilon() {
        System.out.println("G");
    }
}
```

Suppose we declare and initialize these variables (all of these assignments are legal):

```
Two m = new Two();
Top q = new One();
Top r = new Two();
One s = new Two();
```

This problem continues on the next page.

Recall: Polymorphism and Subclasses

Example: class hierarchy using Java inheritance

Recall: Polymorphism and Subclasses

- A subclass instance **is a** superclass instance
  - Polymorphism still works!

```
BankAccount ba = new SavingsAccount();
ba.deposit(100);
```

- But not the other way around!

```
SavingsAccount sa = new BankAccount();
sa.addInterest();
```

- Why not?

BOOM!

Recall: Polymorphism and Subclasses

- A subclass instance **is a** superclass instance
  - Polymorphism still works!

    Superclass = new Subclass(); // legal

    ```
    BankAccount ba = new SavingsAccount();
    ba.deposit(100);
    ```

  - But not the other way around!

    Subclass = new Superclass(); // not legal

    ```
    SavingsAccount sa = new BankAccount();
    sa.addInterest();
    ```

- Why not?

BOOM!

Summary

- If B extends or implements A, we can write

```
A x = new B();
```

Declared type tells which methods x can access. Compile-time error if try to use method not in superclass A, even it is in subclass B.

The actual type tells which class' version of the method to use.

- Can cast to recover methods from B:

```
((B)x).foo()
```

Now we can access all of B's methods too.

If x isn't an instance of B, it gives a run-time error (class cast exception)

Determining Method at Runtime

- Step 1: Identify the Declared/Casted Type
  - This is the item to the left of the variable name where the variable was declared:
    - BankAccount sa = **new** SavingsAccount();

Declared Type

  - Declared Type may be changed a method call due to a cast:
  - ((SavingsAccount)sa).addInterest();

Casted Type

  - If there is a casted type, record that, otherwise use the declared type.

Determining Method at Runtime

- Step 2: Identify the Instantiation/Actual Type
  - This is the type on the right-hand side of the equal sign the last time the variable was assigned to:
    - BankAccount sa = **new** SavingsAccount();

Instantiation Type

  - Record the instantiation type

Determining Method at Runtime

- Step 3 – Part 1: Check for Compilation Errors

  Calling a method that is not available based on the declared or casted type of the object

  ```
  BankAccount sa = new SavingsAccount();
  sa.addInterest();
  ```

  Compiler Error: BankAccount does not have addInterest()

Determining Method at Runtime

- Step 3 – Part 2: Check for Compilation Errors
  - Incompatible type assignment
    - SavingsAccount x = new BankAccount();
    - Compiler Error: BankAccounts can not be stored in SavingAccount typed variables

Determining Method at Runtime

- Step 3 – Part 3: Check for Compilation Errors

  Invalid cast: casting to a type that isn't in the tree below the declaration type.

  BankAccount sa = new SavingsAccount();

  ((SafetyDepositBox)sa).deposit();

  SafetyDepositBox does not inherit from BankAccount

Determining Method at Runtime

- Step 4: Check for Runtime Errors

  Runtime errors are caused by invalid casting.

  An item may only be cast to a type IF:

  - The instantiation type matches the casted type
  - The casted type is between Object and the *instantiation* type

  BankAccount sa = new SavingsAccount();

  ((CheckingAccount)sa).deductFees();

  Runtime Error: SavingsAccount is not a CheckingAccount

Determining Method at Runtime

- Step 4: Check for Runtime Errors

  Account a = new CheckingAccount();

  ((BankAccount)a).deposit();

  This is valid because a CheckingAccount is a BankAccount

  That is, the casted type is between Object and the *instantiation* type

Determining Method at Runtime

- Step 5: Find Method to Run

  Find the instantiation type in the hierarchy.

  1. If that instantiation type implements the given method, then use that implementation.

  2. Otherwise, move up to the parent type and see if there's an implementation there.
     a. If there is an implementation, use that
     b. Otherwise, repeat step 2 until an implementation is found

Exercise

- Do questions 5 through 7 from Quiz.
- Please hand them in when done and then start reading the BallWorlds specification on your schedule page.

Q5-7, hand in when done, then start reading BallWorlds spec

Determing Method at Runtime

- Step 1: Identify the Declared/Casted Type
  - DeclaredType var = …. (can change!)
  - CastedType var2 = (CastedType)var;
- Step 2: Identify the Instantiation/Actual Type
  - (Never changes! To the right of original "= new *")
  - …= new InstantiatedType();
- Step 3: Check for Compilation Errors
  - Method not available based on declared/casted type
  - Incompatible type assignment: Dog x = new Cat()
  - Invalid cast: casting to type not below declaration type
- Step 4: Check for Runtime Errors
  - Instantiation type must matches casted type OR
  - Casted type is between declaration and instantiation type
- Step 5: Find Method to Run
  - Start with instantiation type and look to super class!