Polymorphism

Introduction to Programming and Computational Problem Solving II

CSE 8B

Lecture 13

Announcements

- Assignment 5 is due today, 11:59 PM
  - Upgrade beginning Feb 27, 12:01 AM
- Assignment 6 will be released Feb 26
  - Due Mar 5, 11:59 PM

Inheritance

- Inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)
- A subclass inherits from a superclass
  - For example, both a circle and a rectangle are geometric objects
    - `GeometricObject` is a superclass
    - `Circle` is a subclass of `GeometricObject`
    - `Rectangle` is a subclass of `GeometricObject`
- Models **is-a** relationships
  - For example
    - Circle **is-a** GeometricObject
    - Rectangle **is-a** GeometricObject

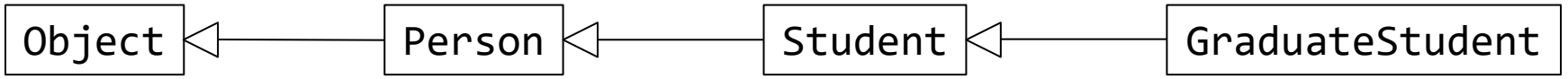
Polymorphism

- Remember, a class defines a type
- A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*
  - For example
    - `Circle` is a subtype of `GeometricObject`, and `GeometricObject` is a supertype for `Circle`
- *Polymorphism* means that a variable of a supertype can refer to a subtype object
  - Greek word meaning “many forms”

Polymorphism

- An object of a *subtype* can be used wherever its *supertype* value is required
  - For example
    - Method `m` takes a parameter of the `Object` type, so you can invoke it with any object

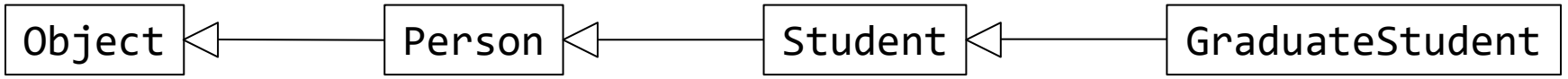
```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
}

class Person {
}
```



Declared type and actual type

- The type that declares a variable is called the variable's *declared type*
- The actual class for the object referenced by the variable is called the *actual type* of the variable
- Remember, a variable of a reference type can hold a `null` value or a reference to an instance of the declared type

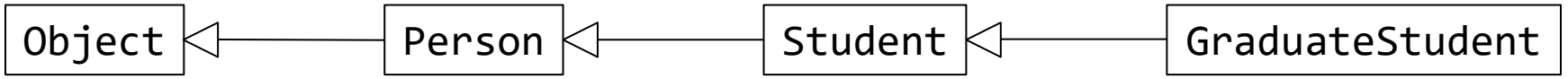
```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
}

class Person {
}
```



Declared type and actual type

- In all executions of `m`, the variable `x`'s *declared type* is `Object`
- In the first execution of `m`, the variable `x`'s *actual type* is `GraduateStudent`
- In the second execution of `m`, the variable `x`'s *actual type* is `Student`
- In the third execution of `m`, the variable `x`'s *actual type* is `Person`
- In the fourth execution of `m`, the variable `x`'s *actual type* is `Object`

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
}

class Person {
}
```

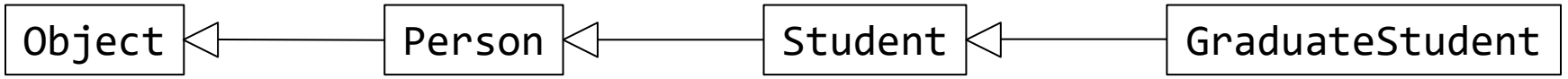


Dynamic binding

- When the method `m` is executed, the argument `x`'s `toString` method is invoked
- `x` may be a reference to an instance of `GraduateStudent`, `Student`, `Person`, or `Object`
- Classes `Student`, `Person`, and `Object` have their own implementation of the `toString` method
- Which implementation is used will be determined dynamically by the JVM at *runtime*
- This capability is known as *dynamic binding*

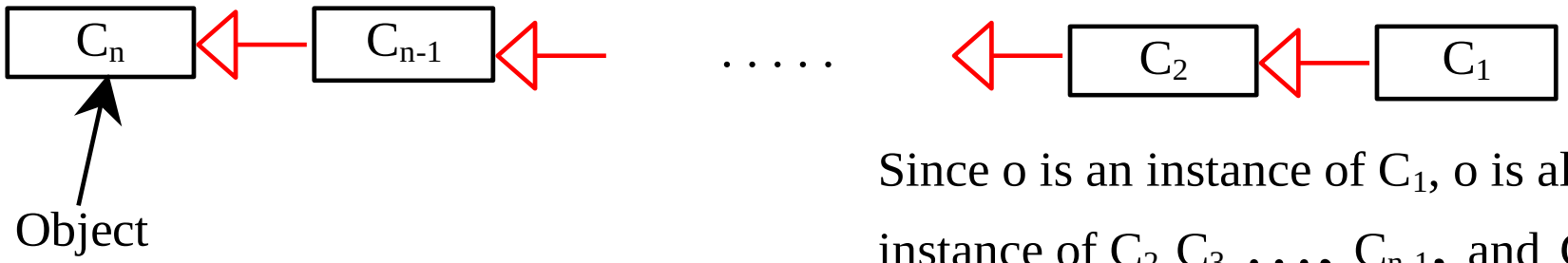
```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person {
  public String toString() {
    return "Person";
  }
}
```

Method overridden in subclasses



Dynamic binding

- Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n
  - That is, C_n is the most general class, and C_1 is the most specific class
- In Java, C_n is the Object class
- If object o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, **in this order**, until it is found
- Once an implementation is found, the search stops and the first-found implementation is invoked



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . ., C_{n-1}, and C_n

Matching and binding

- *Matching* a method *signature*
  - The *declared type* of the reference variable decides which method to match at *compile time*
- *Binding* a method *implementation*
  - A method may be implemented in several classes along the inheritance chain
  - The *actual type* of the reference variable decides which implementation of the method the JVM dynamically binds at *runtime*

Matching and binding

- In all executions of `m`, the variable `x`'s *declared type* is `Object`
- In the first execution of `m`, the variable `x`'s *actual type* is `GraduateStudent`
- In the second execution of `m`, the variable `x`'s *actual type* is `Student`
- In the third execution of `m`, the variable `x`'s *actual type* is `Person`
- In the fourth execution of `m`, the variable `x`'s *actual type* is `Object`

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person {
  public String toString() {
    return "Person";
  }
}
```

Matching at compile time

Method overridden in subclasses

Binding at runtime

Casting objects

- You have been using the casting operator to convert variables of one primitive type to another
- Casting can also be used to convert an object of one class type to another within an inheritance hierarchy
  - This is called *casting objects*

Upcasting is implicit

- The statement

```
m(new Student());
```

  is equivalent to

```
Object o = new Student();
m(o);
```

Implicit casting

- It is always possible to cast an instance of a subclass to a variable of a superclass
  - This is called *upcasting*

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
}

class Person extends Object {
}
```

Downcasting

- **Warning: if you find yourself wanting to perform (explicit) downcasting from a superclass to a subclass, it is a sign you are likely approaching things the wrong way!**
- Override methods in subclasses instead

Downcasting

- **Downcasting is such a bad practice** that explicit casting must be used to confirm your intention to the compiler
- For example

```
Object o = new Student();
m(o);
Student b = o; // Compile error
Student c = (Student)o; // No error
```

Downcasting

- If you are downcasting a superclass object to an object that is not an instance of a subclass, then a runtime exception occurs
- Use the `instanceof` operator to avoid this
  - For example

```
void someMethod(Object myObject) {
... // Some lines of code
// Perform casting if myObject is an instance of Circle
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ... // Some lines of code
}
```

"Safe" downcasting

Explicit casting

Override `equals` method in `Object`

- Remember, usually a class should override the `toString` method so it returns a digestible string representation of the object
- You may also want to override the `equals` method
  - One of the few reasonable times to use downcasting
  - Always override `hashCode` when you override `equals`

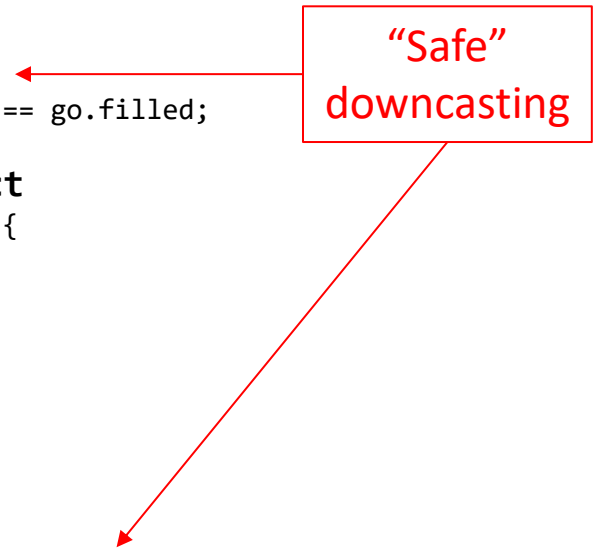
Override `equals` method in `Object` using **polymorphism**

- For example, **class extending `Object`**

```
public class GeometricObject {
  private String color;
  private boolean filled;
  ...

  @Override
  public boolean equals(Object o) {
    if (o == this)
      return true;
    if (!(o instanceof GeometricObject))
      return false;
    GeometricObject go = (GeometricObject)o;
    return color.equals(go.color) && filled == go.filled;
  }
```

- and **class extending any class other than `Object`**

```
public class Circle extends GeometricObject {
  private double radius;
  ...

  @Override
  public boolean equals(Object o) {
    if (o == this)
      return true
    if (!(o instanceof Circle))
      return false;
    return super.equals(o) && radius == ((Circle)o).radius;
  }
```



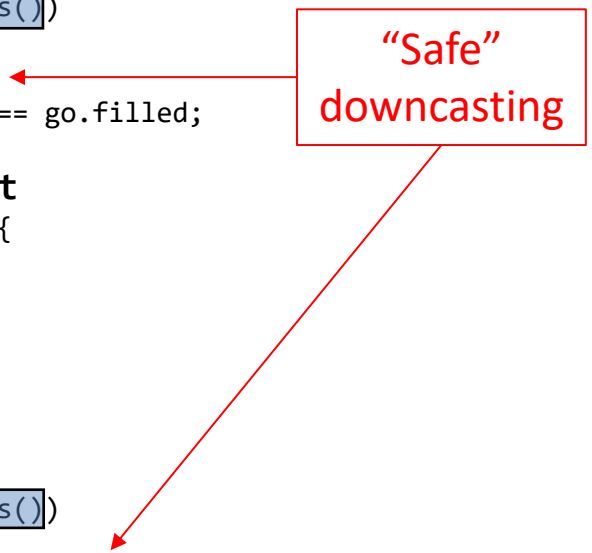
Override `equals` method in `Object` using **strict class equality**

- For example, **class extending `Object`**

```
public class GeometricObject {
  private String color;
  private boolean filled;
  ...

  @Override
  public boolean equals(Object o) {
    if (o == this)
      return true;
    if (o == null || getClass() != o.getClass())
      return false;
    GeometricObject go = (GeometricObject)o;
    return color.equals(go.color) && filled == go.filled;
  }
```

- and **class extending any class other than `Object`**

```
public class Circle extends GeometricObject {
  private double radius;
  ...

  @Override
  public boolean equals(Object o) {
    if (o == this)
      return true
    if (o == null || getClass() != o.getClass())
      return false;
    return super.equals(o) && radius == ((Circle)o).radius;
  }
```



Advanced topic

Always override hashCode when you override equals

- **Equal objects must have equal hash codes**

- For example, **class extending `Object`**

```
public class GeometricObject {
  private String color;
  private boolean filled;
  ...

  @Override
  public int hashCode() {
    int result = color == null ? 0 : color.hashCode();
    result = 31 * result + Boolean.hashCode(filled);
    return result;
  }
```

- and **class extending any class other than `Object`**

```
public class Circle extends GeometricObject {
  private double radius;
  ...

  @Override
  public int hashCode() {
    int result = super.hashCode();
    result = 31 * result + Double.hashCode(radius);
    return result;
  }
```

Fast and fair approximation of ideal hash function. Note 31 is an odd prime.

Object reference variable

Primitive type

For arrays, use Arrays.hashCode

Methods and data fields visibility

Covered later in the quarter

Modifiers on Members in a Class	Accessed from Same Class	Accessed from Any Class in Same Package	Accessed from Any Class in Same Package and Any Subclass in Any Package	Accessed from Any Class in Any Package
Public	✓	✓	✓	✓
Protected	✓	✓	✓	
Default (no modifier)	✓	✓		
Private	✓			

Subclass and visibility/accessibility

- If desired, a subclass **can increase accessibility** of a method defined in the superclass, but a subclass **cannot decrease accessibility** of a method defined in the superclass
  - For example, a subclass may override a `protected` method in its superclass and change its visibility to `public`
  - For example, if a method is defined as `public` in the superclass, it must be defined as `public` in the subclass

Preventing extending and overriding

- You may occasionally want to prevent classes from being extended
- In such cases, use the `final` modifier to indicate a class is final and cannot be a parent class

The `final` modifier

- A `final` class cannot be extended
  - For example

```
public final class Math {
   ...
}
```

- A `final` method cannot be overridden by its subclasses
- And remember, a `final` variable is a constant
  - For example

```
public static final double PI = 3.14159;
```

The `final` modifier

- Modifiers are used on classes and class members (data and methods), except the `final` modifier can also be used on local variables in a method
- A `final` local variable is a constant inside a method
- **A best practice is to use `final` variables liberally**

Modifiers

- Access modifiers
  - For classes
    - `public` and default (no modifier)
  - For methods (*including* constructors) and data fields
    - `public`, `protected`, default (no modifier), and `private`
- Non-access modifiers
  - For classes
    - `final` and `abstract` (covered later in the quarter)
  - For methods (*excluding* constructors)
    - `final`, `static`, and `abstract` (covered later in the quarter)
  - For data fields
    - `final` and `static`
- All modifiers
  - https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html
  - https://docs.oracle.com/javase/specs/jls/se11/html/jls-8.html

Next Lecture

- Exceptions
- Text file input/output