

Notes

Project01 Part 2 due Friday

- Please sign up for your interview if you have not yet done so
- Bring a WRITTEN description of how you tested
- Please fix your problems from Part 1—get help if you need it

Tournament Wednesday April 10!

- 7pm, Lo Schiavo 307
- Please RSVP!

In-class Lab today

Thurs office hours delayed (again), to 1-2:30pm

REVIEW: Class Relationships

Classes in a software system can have various types of relationships to each other

Three of the most common relationships:

- Inheritance: A *is-a* B
- Dependency: A *uses* B
- Aggregation: A *has-a* B

When you design several classes that interact, **understand the relationships** to choose whether to use inheritance, dependency, or containment.

Copyright © 2014 Pearson
Education, Inc.

Student is-a Person

Inventory uses Math

Student has-a String (name, address, etc)

Advanced Topics in Class Design

Spring 2024

Copyright 2023 Paul Haskell. All rights reserved.

Member classes

Inheritance

Overriding Inherited Methods

Protected variables and methods

class Object

static variables and methods

Class relationships

Polymorphism

Binding

Consider the following method invocation:

```
obj.doIt();
```

The `doIt()` method is *bound* to the object that invokes it

If this binding occurred at compile time, then that line of code would call the same method every time

However, Java defers method binding until run time -- this is called *dynamic binding* or *late binding*

Polymorphism

The term *polymorphism* literally means "having many forms"

A *polymorphic reference* is a variable that can refer to different types of objects at different points in time

A method called through a polymorphic reference can change from one invocation to the next, depending on the actual type of the underlying object

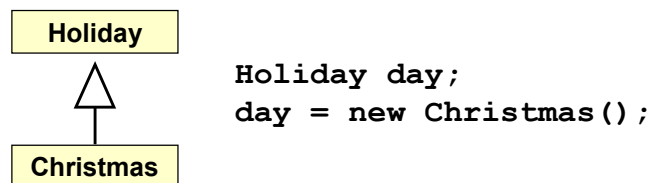
Java references are polymorphic

References and Inheritance

An object reference can refer to an object...

- whose type is the type of the reference, or
- whose type is any class derived from it by inheritance

This is the meaning of the "is-a" relationship!



Polymorphism

```
public class Person {  
    String toString() { return new String("I am a Person"); }  
}  
  
public class Adult extends Person {  
    String toString() { return new String("I am an adult"); }  
}  
  
public class CollegeStudent extends Adult {  
    String toString() { return new String("I am a college student"); }  
}  
  
...  
  
void Identify(Person p) { System.out.println(p); }
```

Polymorphism

```
public static void main(String[] args) {  
    Person people[] = new Person[3];  
    people[0] = new Person();  
    people[1] = new Adult();  
    people[2] = new CollegeStudent();  
    for(int n = 0; n < 3; n++) {  
        Identify(people[n]);  
    }  
}
```

LET'S UNPACK THIS. This prints the obvious. But HOW is important.

The Identify() function does not know about anything but Person. It just calls the toString() method on each object

Each object knows for itself which version of toString() it should use

VERY POWERFUL!

Users of a base class do not have to know details about derived classes! As long as derived classes do the right thing!

INHERITANCE V USEFUL IN SOME CIRCUMSTANCES. DON'T USE IT IF IT ISN'T NEEDED!

Quick Check

If `MusicPlayer` is the parent of `MP3Player`, are the following assignments valid?

```
MusicPlayer mplayer = new MP3Player();
```

```
MP3Player mp3player = new MusicPlayer();
```

Advanced Topics in Class Design

Member classes

Inheritance

Overriding Inherited Methods

Protected variables and methods

class Object

static variables and methods

Class relationships

Polymorphism

Multiple inheritance, Abstract classes, and Interfaces

Multiple Inheritance

Java supports *single inheritance*, meaning that a derived class can have only one parent class

Multiple inheritance allows a class to be derived from two or more classes, inheriting the members of all parents

Copyright © 2014 Pearson
Education, Inc.

C++ allows multiple inheritance.

INTERESTING JAVA DESIGN PHILOSOPHY. WALK THRU NAMESPACE COLLISION PROBLEM

Abstract Classes

An *abstract method* is a method header without a method body

- **Declares** function prototype but does not **define** function statements

```
abstract String status();
```

An abstract method must be in an "abstract class"

```
abstract class Machine { ... }
```

We cannot construct objects of an Abstract class...but we can derive from it!

Copyright © 2014 Pearson
Education, Inc.

```
Abstract class Animal { abstract String makeSound(); }
```

```
Abstract class Cow extends Animal { String makeSound() { return "moo!"; } }
```

Interfaces

A Java *interface* is a class, all of whose methods are abstract
(and all of whose variables, if any, are `final`)

An *interface* is used to establish (guarantee) that a child class will
implement the *interface's* set of (abstract) methods

Interfaces

Since there is no danger of namespace collision with interface parents, Java allows inheritance from multiple interfaces

- Use `inherits` keyword

Interface Hierarchies

Inheritance can be applied to interfaces

That is, one interface can be derived from another interface

The child interface inherits all abstract methods of the parent

A class implementing the child interface must define all methods from both interfaces

Interfaces

interface is a reserved word

None of the methods in
an interface are given
a definition (body)

```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2(double value, char ch);
    public boolean doTheOther(int num);
}
```

A semicolon immediately
follows each method header

Interfaces

An interface or abstract class cannot be instantiated

Methods in an interface have public visibility by default


A class formally implements an interface by:

- stating so in the class header
- providing implementations for every abstract method in the interface

If a class implements an interface, it must define *all* methods in the interface

Interfaces

implements is a
reserved word



```
public class CanDo implements Doable
{
    public void doThis()
    {
        // whatever
    }

    public void doThat()
    {
        // whatever
    }

    // etc.
}
```

Each method listed
in Doable is
given a definition

Polymorphism via Interfaces

Interfaces can be used to set up polymorphism!

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
    public void speak();
    public void announce(String str);
}
```

Polymorphism via Interfaces

An interface name can be used as the type of an object reference variable:

```
Speaker current;
```

The `current` reference can be used to point to any object of any class that implements the `Speaker` interface

The version of `speak` invoked by the following line depends on the type of object that `current` is referencing:

```
current.speak();
```

Polymorphism via Interfaces

Now suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method

In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philosopher();  
guest.speak();  
guest = new Dog();  
guest.speak();
```

Polymorphism via Interfaces

As with class reference types, the compiler will restrict invocations to methods in the interface

For example, even if `Philosopher` also had a method called `pontificate`, the following would still cause a compiler error:

```
Speaker special = new Philospher();  
special.pontificate(); // compiler error
```

Remember, the compiler bases its rulings on the type of the reference, not the type of the actual object

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();  
Philosopher second = new Philosopher();  
second.pontificate();  
first = second;
```

Yes, all assignments and method calls are valid as written

Abstract Classes - Details

An abstract class often contains abstract methods with no definitions (like an interface)

Unlike with an `interface`, the `abstract` modifier must be applied to each abstract method

An abstract class typically also contains non-abstract methods with full definitions

A class declared as `abstract` does not have to contain abstract methods -- simply declaring it as `abstract` makes it so

Copyright © 2014 Pearson
Education, Inc.

LOOK AT EXAMPLES! `ReadableExample.java`

Designing for Inheritance

Taking the time to create a good software design reaps long-term benefits

Inheritance issues are an important part of an object-oriented design

Properly designed inheritance relationships can contribute greatly to the elegance, **maintainability**, and **reuse** of the software

Inheritance Design Guidelines

Every inheritance should be an is-a relationship

- A Dictionary "is-a" Book. A Student "is-a" Person.

Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

Find common characteristics of classes and push them as high in the class hierarchy as appropriate

Override methods as appropriate to tailor or change the functionality of a child

Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Guidelines

Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data

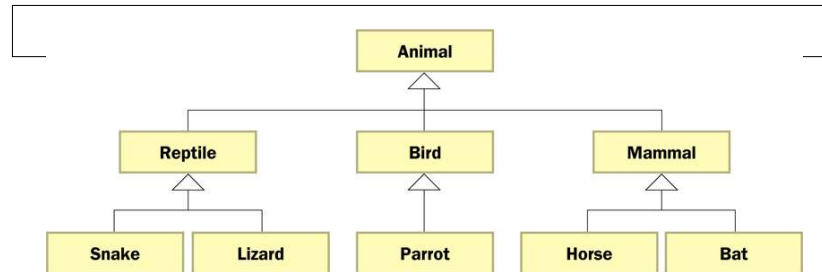
Override general methods such as `toString()` and `equals()` with appropriate definitions

Use abstract classes and interfaces to represent general concepts that derived classes have in common but that don't have a reasonable "default" behavior

Use visibility modifiers carefully to provide needed access without violating encapsulation

Class Hierarchies

A child class of one parent can be the parent of another child, forming a *class hierarchy*



The Exception Class Hierarchy

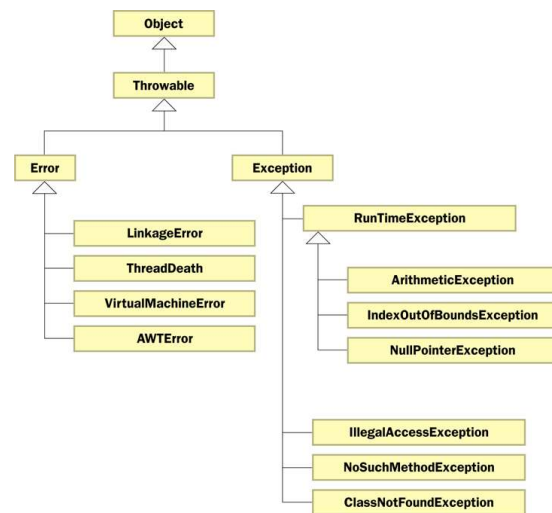
Exception classes in the Java API are related by inheritance, forming an exception class hierarchy

All **error** and **exception** classes are descendants of the `Throwable` class

A programmer can define an exception by extending the `Exception` class or one of its descendants

The parent class used depends on how the new exception will be used

The Exception Class Hierarchy



Copyright © 2014 Pearson
Education, Inc.

ERRORS SHOULD NOT BE HANDLED. Philosophy is to let the program fail. If you were flying the space shuttle, you'd be a little nervous if the life support system were running Java, no?

Checked Exceptions

An exception is either *checked* or *unchecked*

A *checked exception* must either be caught or must be listed in the `throws` clause of any method that may throw or propagate it

The compiler will issue an error if a checked exception is not caught or listed in a `throws` clause

Unchecked Exceptions

An unchecked exception does not require explicit handling, though it could be processed that way

The only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants

Errors are similar to `RuntimeException` and its descendants in that:

- Errors should not be caught
- Errors do not require a `throws` clause

Quick Check

Which of these exceptions are checked and which are unchecked?

<code>NullPointerException</code>	<code>Unchecked</code>
<code>IndexOutOfBoundsException</code>	<code>Unchecked</code>
<code>IOException</code>	<code>Checked</code>
<code>ArithmeticException</code>	<code>Unchecked</code>

One more
thing...