CSE 142

Inheritance In Java

Outline for Today

- Review from last time
  - Classes can be related by ***inheritance***
    - Defines an "is-a" relationship between classes
  - A ***base*** class (or ***superclass***) defines properties/ responsibilities shared by a set of related classes
  - A ***derived*** class (or ***subclass***) extends a base class
    - *Inherits* all of the properties/responsibilities of the base class
    - Can define additional properties/responsibilities
- Goal for today
  - Learn how to do this in Java
  - Explore some of the implications

Library Circulation System Classes

- Simplified version of design from last time
- Class CirculationItem – class with common information
  - State: title, call number, and whether checked out
  - Methods: retrieve title, call number; check in and out, etc.
- Class Book – extended version of CirculationItem
  - Additional state – author
  - Additional methods – get author
- Class Journal – extended version of CirculationItem
  - Additional state – list of articles
  - Additional methods – get/set list of articles

Class CirculationItem (1)

- Very similar to other classes we've seen

```
/** Representation of common properties of items in a library's circulation system */
public class CirculationItem {
  // instance variables
  private String  title;                  // book or journal title
  private String  callNumber;             // Library of Congress call #
  private boolean checkedOut;             // = "this item is currently checked out"

  /** Construct new CirculationItem with specified title and call # */
  public CirculationItem(String title, String callNumber) { … }}

  /** Return the title of this CirculationItem */
  public String getTitle() { …  }

  /** Return the call number of this CirculationItem */
  public String getCallNumber() { …  }
  …
```

Class CirculationItem (2)

```
...
 /** Check in this CirculationItem */
 public void checkin() {  checkedOut = false;  }

 /** Check out this CirculationItem */
 public void checkout() {  checkedOut = true;  }

 /** = "this item is in the library" */
 public boolean inLibrary() {  return !checkedOut;  }

 public String toString() {
   return "CirculationItem(title=" + title + ", callNumber=" + callNumber +
      "checked out=" + checkedOut + ")";
 }
}
```

Class Book

- **Like a regular class declaration, but with an *extends* clause**

```
/** Representation of a book */
public class Book extends CirculationItem {
    // additional state
    private String author;        // author(s) of this book
    /** Construct a Book with the given title, author, and call number */
    public Book(String title, String author, String callNumber) { … }
    /** return the author of this book */
    public String getAuthor( ) { return author; }
    /** return a string representation of this book */
    public String toString( ) { … }
}
```

Implications

- **A Book object *is a* CirculationItem with additional state (author) and methods (constructor, getAuthor)**
- **Each instance of Book contains all of the state inherited from CirculationItem plus the additional state declared in Book**
  - **But private information in CirculationItem is accessible only inside that class – something we'll have to deal with**
- **Any method in either class can be applied to an instance of Book**
  - **Has to be visible (public) at the point it is used, of course**

Draw the Diagram

```
Book tome = new Book("War and Peace", "Tolstoy", "PG3366.V6 1991");
```

Constructing a Book

- **When we define a class, we should use constructors to properly initialize the state of instances of the class**
- **For a Book…**
  - **trivial to initialize author instance variable**
  - **How do we initialize the inherited state (title, call #, checkedOut)?**
    - Can't reference inherited fields directly – they're private (and we want them to stay that way)
    - Can't "call" a constructor and don't want a new, separate, CirculationItem object
  - **Solution: special syntax to run superclass (CirculationItem) constructor at very beginning of Book constructor**

      super(*arguments*);

    Must be the very first thing in the Book constructor

Book Constructor

```
/** Construct a book with the given title, author, and call number */
public Book(String title, String author, String callNumber) {








}
```

Execution – Yet Another Diagram

- **What really happens when we execute**

  Book tome = new Book("War and Peace", "Tolstoy", "PG3366.V6 1991");

Using Books

- **Demo**

```
Book b = new Book(…)
b.getAuthor()
b.checkout()
…
```

Books and CirculationItems (1)

- Book and CirculationItem are both types
- An instance of Book has type Book…
- … and *also* has type CirculationItem (since a Book is an extended CirculationItem)
  - So this works

```
Book b = new Book( … );
CirculationItem c = b;
c.getTitle( )
```

Books and CirculationItems (2)

- But neither of these are allowed, even though CirculationItem c actually refers to an instance of Book (why not?)

```
c.getAuthor( )
Book novel = c;
```

- Solution: we can use a cast to claim that it really is a Book (checked at runtime and trouble if it isn't)

Another Extended Class – Journal

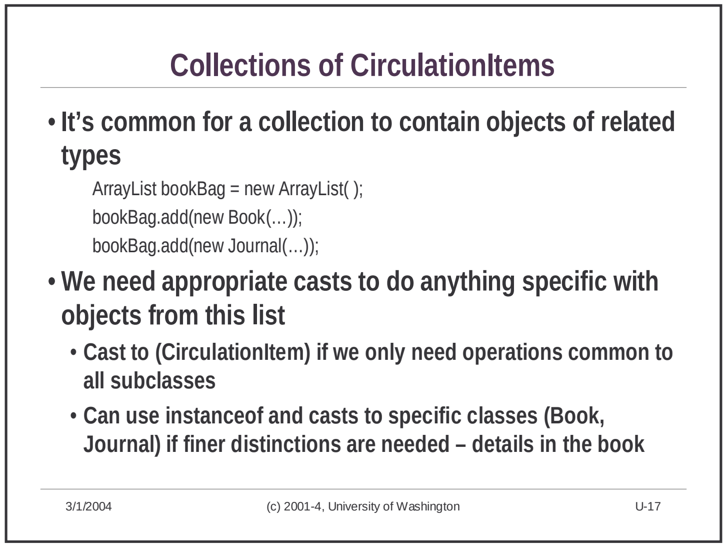
```
/** Representation of a Journal */
public class Journal extends CirculationItem {
    // additional state
    private ArrayList articles;        // names of articles in this Journal
    /** Construct a new Journal with title, call number, and empty article list */
    public Journal(String title, string callNumber) {




    }
    // additional methods to get and set article list omitted
}
```

Mixing Journals and Books

- Since Books and Journals are all CirculationItems, we can write methods that can process any of these without having to distinguish which one – as long as we only use methods defined in CirculationItem

```
/** print the title of the given CirculationItem */
public void printTitle(CirculationItem item) {
    System.out.println(item.getTitle());
}
```

  - Same idea when using interface types as parameters
  - Method printTitle is said to be ***polymorphic*** (meaning many types) because its parameters can be objects of different related types



Collections of CirculationItems

- It's common for a collection to contain objects of related types

```
ArrayList bookBag = new ArrayList( );
bookBag.add(new Book(…));
bookBag.add(new Journal(…));
```

- We need appropriate casts to do anything specific with objects from this list
  - Cast to (CirculationItem) if we only need operations common to all subclasses
  - Can use instanceof and casts to specific classes (Book, Journal) if finer distinctions are needed – details in the book