

CSE 142

Inheritance: Types, Classes, and Methods

1/10/2003

(c) 2001-3, University of Washington

V-1

Outline for Today

- Review
 - Basic ideas of inheritance
 - Types, classes, and objects
- Goal for today
 - Look at details of inheritance more closely
 - Method overriding and overloading
 - Class Object

1/10/2003

(c) 2001-3, University of Washington

V-2

From Last Time...

- Library Circulation system
- 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

1/10/2003

(c) 2001-3, University of Washington

V-3

Types (Review)

- Everything in Java has a type
 - A combination of state and operations
- Primitive Types: `int`, `double`, `char`, `boolean`, ...
 - Simple, atomic state
 - Operations built in to Java language: `+`, `-`, `*`, `/`, `%`, `&&`, `||`, `!`, ...
- All other types – references to objects (class instances): `Rectangle`, `Color`, `Pixel`, `CirculationItem`, `Book`, ...
 - State is collection of instance variables
 - Operations are methods
- Each class definition specifies a new type with that name

1/10/2003

(c) 2001-3, University of Washington

V-4

Types and Inheritance (1)

- When we define

```
class Book extends CirculationItem { ... }
```

we create a new type, `Book`
- Instances of class `Book` have type `Book`, and also...
- ...have type `CirculationItem`
 - Not so odd if you think about it. Many things in the real world have multiple “types” or roles. A person can be a student, employee, partner, child, parent,

1/10/2003

(c) 2001-3, University of Washington

V-5

Types and Inheritance (2)

- ```
class Book extends CirculationItem { ... }
```
- Rule: every `Book` object is also a `CirculationItem` object
  - Can be used in any situation where either a `Book` or `CirculationItem` is expected

```
Book b = new Book(...);
Book x = b;
CirculationItem c = b;
```
- The reverse is not true: there are `CirculationItem`s that are not `Books` (plain `CirculationItem`s, `Journals`)
  - So this is not allowed

```
CirculationItem c = new CirculationItem(...); // ok
Book b1 = c; // compile-time type error
Book b2 = (Book) c; // run time class cast exception error
```

1/10/2003

(c) 2001-3, University of Washington

V-6

## Dynamic and Static Types

- The **static type** of a variable is the type in its declaration

```
Book b = ...
Journal m = ...
CirculationItem c = ...
```

- The **dynamic type** of a variable is the type of the object it currently refers to
  - Either the variable's static type or a type that extends it
  - Can change during execution

1/10/2003

(c) 2001-3, University of Washington

V-7

## Dynamic Types

- What are the dynamic types of the variables in the following code?

```
Book b = new Book("Short Story", "A. U. Thor", "P34.56");

CirculationItem c = new CirculationItem("Rather Bland", "A1");

CirculationItem d = new Journal("Long 'n Boring", "O45.367");

c = b;
```

1/10/2003

(c) 2001-3, University of Washington

V-8

## Static Types and Methods

- If we declare a variable

```
CirculationItem c = ...
```

the only guarantee we have is that it refers to some sort of **CirculationItem**

- Compiler doesn't attempt to trace values assigned to variables to decide type information
- So the only methods we can call using the variable **c** are the ones available in its static type (**CirculationItem**)

1/10/2003

(c) 2001-3, University of Washington

V-9

## Example

- The following produces a compile-time type error

```
Book b = new Book("Exciting", "Great Author", "H396.47");
CirculationItem c = b; // fine
System.out.println(c.getAuthor()); // no - static type of c doesn't include
// a getAuthor() method
```

- But if we're sure it will really be a **Book** at runtime, we can use a cast

```
Book temp = (Book)c; // ok
System.out.println(temp.getAuthor()); // fine - temp is a Book
```

or

```
System.out.println(((Book) c).getAuthor()); // also ok
```

1/10/2003

(c) 2001-3, University of Washington

V-10

## toString()

- So what's the story with **toString()**?

- All three classes (**CirculationItem**, **Book**, **Journal**) contain one of these
- How do we decide which one to use?

```
Book b = new Book(...);
CirculationItem c = b;
System.out.println(c); // CirculationItem toString() or Book toString()?
```

1/10/2003

(c) 2001-3, University of Washington

V-11

## Method Override and Dynamic Dispatch

- When we extend a class, we can redefine a method that we would otherwise inherit from the original class
- The redefined method is said to **override** the original method definition
- When we call a method, the **dynamic type** of the object is used to select the appropriate method

```
CirculationItem c = new Book(...);
System.out.println(c); // dynamic type of c here is Book, so
// toString() from Book is used
```

- This is called **dynamic (method) dispatch**

1/10/2003

(c) 2001-3, University of Washington

V-12

## Dynamic Dispatch and Class Hierarchy Design

- Overriding and dynamic dispatch are powerful design tools
- Idea: when designing a class hierarchy, define in the original class methods which we want to be available for all objects in the hierarchy
- Use overriding to provide specialized implementations in extended classes
- Dynamic dispatch guarantees that the appropriate overriding methods will be called

1/10/2003

(c) 2001-3, University of Washington

V-13

## Class Object

- The Java class structure has a root class: **Object**
- All Java classes implicitly extend **Object** if they don't explicitly extend some other class (which itself extends **Object** directly or indirectly)

```
class CirculationItem { ... }
```

means exactly the same thing as

```
class CirculationItem extends Object { ... }
```

- Classes like **ArrayList** have parameters and results of type **Object**, so will handle any non-primitive type

```
public void add(Object obj) { ... }
```

```
public Object get(int position) { ... }
```

1/10/2003

(c) 2001-3, University of Washington

V-14

## What's in Class Object?

- Object contains methods (not many) that are suitable for all classes
- Class definitions can override these to provide more appropriate, specific versions
- Examples we've seen frequently
  - `toString()`
  - `equals()`

1/10/2003

(c) 2001-3, University of Washington

V-15

## Overloading

- In a class, it is possible to define more than one method with the same name

```
class Thing {
 /** do something interesting with a Rectangle */
 public void doIt(Rectangle r) { ... }
 /** do something interesting with an int */
 public void doIt(int n) { ... }
}
```

- This is called method **overloading**
  - **Not** the same thing as method overriding  
(overriding is substituting a new method for one that would otherwise be inherited when we extend a class)
- Compiler picks right method to use by comparing call argument types with parameters of available methods

1/10/2003

(c) 2001-3, University of Washington

V-16

## Example of Overloading – `System.out.println`

- We've been able to use `System.out.println` to print anything. How does this work?
- Answer: this method is overloaded for all the basic types and for class **Object**

```
System.out.println(int)
System.out.println(double)
System.out.println(char)
System.out.println(boolean)
System.out.println(Object) // uses toString() to get string to be printed -
... // works for every kind of object (why?)
```

- Compiler picks actual method to use depending on type of thing being printed

1/10/2003

(c) 2001-3, University of Washington

V-17

## That's (almost) It!

- Key ideas
  - Class definition by extension ("is-a")
  - Inheritance
  - Static and dynamic types
  - Method overriding
  - Dynamic dispatch
  - Method overloading
  - Class **Object**
- Still to do
  - Abstract classes
  - Interfaces

1/10/2003

(c) 2001-3, University of Washington

V-18