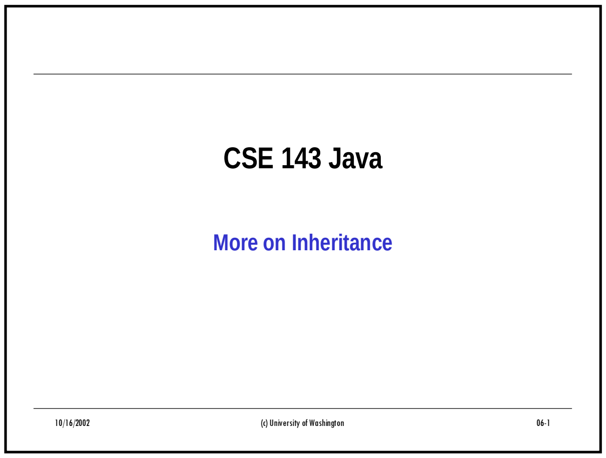
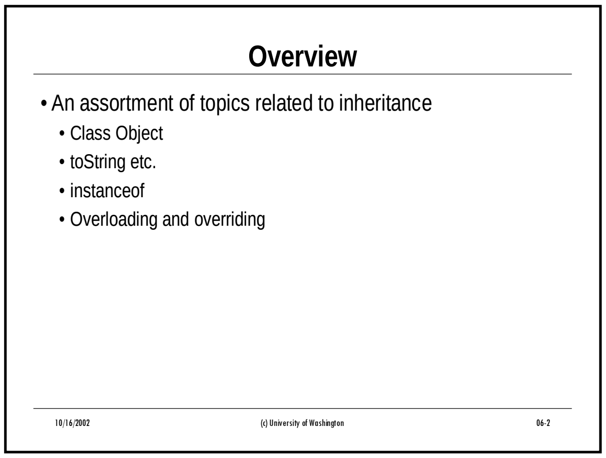


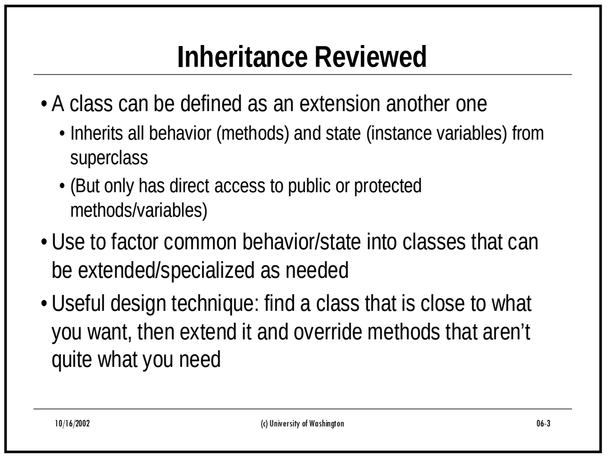
CSE 143 Java

More on Inheritance



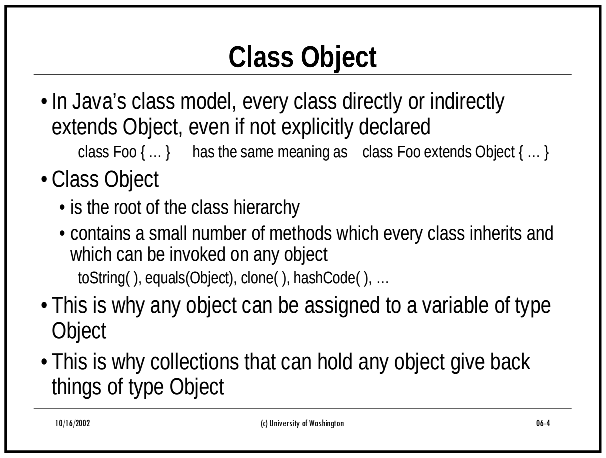
Overview

- An assortment of topics related to inheritance
  - Class Object
  - toString etc.
  - instanceof
  - Overloading and overriding



Inheritance Reviewed

- A class can be defined as an extension another one
  - Inherits all behavior (methods) and state (instance variables) from superclass
  - (But only has direct access to public or protected methods/variables)
- Use to factor common behavior/state into classes that can be extended/specialized as needed
- Useful design technique: find a class that is close to what you want, then extend it and override methods that aren't quite what you need



Class Object

- In Java's class model, every class directly or indirectly extends Object, even if not explicitly declared

  class Foo { … }   has the same meaning as   class Foo extends Object { … }
- Class Object
  - is the root of the class hierarchy
  - contains a small number of methods which every class inherits and which can be invoked on any object

    toString( ), equals(Object), clone( ), hashCode( ), …
- This is why any object can be assigned to a variable of type Object
- This is why collections that can hold any object give back things of type Object

toString( )

- Object provides a default implementation of toString( )

```
MyClass#2376ac65
```

- Most well-designed classes should override toString( ) to return a more useful description of an instance

```
Rectangle[height: 10; width: 20; x: 140; y: 300]
Color[red: 120; green: 60; blue: 240]
(BankAccount: owner=Bill Gates, Balance = beyond your imagination)
```

- Called by many system methods whenever a printable version of an object is needed
- Use with System.out as a debugging tool

```
System.out.println(unusualBankAccount);
System.out.println(suspectRectangle);
```

equals( ), clone( )

- *obj1*.equals(*obj2*) should return true if *obj1* and *obj2* represent the same value
  - Object's implementation just compares objects for identity, using ==
  - If a subclass can have different objects that should act as equal (e.g. Set objects, or Point objects), then it should override equals( )
- *obj*.clone( ) should return a copy of *obj* with the same value
  - Object's implementation just makes a new instance of the same class whose instance variables have the same values as *obj*
  - If a subclass needs to do something different, e.g. clone some of the instance variables too, then it should override clone( )

instanceof (skip for now)

- The expression

```
<object> instanceof <classOrInterface>
```

  is true if the object is an instance of the given class or interface (or any subclass of the one given)
- One common use: checking types of generic objects before casting

```
Monster m = ...;
if (m instanceof JumpingMonster) {
    JumpingMonster jm = (JumpingMonster) m;
    jm.jump(veryHigh);
}
```

- Often can be replaced by method override and dynamic dispatch

```
Monster m = ...;
m.jumpIfPossible(veryHigh);   // Monster does nothing, JumpingMonster overrides to jump
```

Overriding and Overloading

- In spite of the similar names, these are very different
- **Overriding**: replacing an inherited method in a subclass

```
class One {
    public int method(String arg1, double arg2) { … }
}
class Two extends One {
    public int method(String arg1, double arg2) { … }
}
```

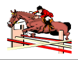
  - Argument lists and results must match **exactly** (number and types)
  - Method called depends on actual (dynamic) type of the receiver

Overloading

- **Overloading**: a class may contain multiple definitions for constructors or methods with the same name, but different argument lists

```
class Many {
    public Many( ) { … }
    public Many(int x) { … }
    public Many(double x, String s) { … }
    public void another(Many m, String s) { … }
    public int another(String[ ] names) { … }
```

  - Parameter lists must differ in number and/or type of parameters
    - Result types can differ, or not
  - Method calls are resolved automatically depending on number and (static) types of arguments – must be a unique best match



Overriding vs Overloading

- Overriding
  - Allows subclasses to substitute an alternative implementation of an inherited method
  - Client still only sees one operation in the class's interface
- Overloading
  - Allows several different methods to (for convenience) have the same name
  - These are **completely independent** of each other; they could have been given different names just as easily
  - Client sees all of the overloaded methods in the class's interface
- One is static, one is dynamic: which is which??
- Can be mixed, but please don't!