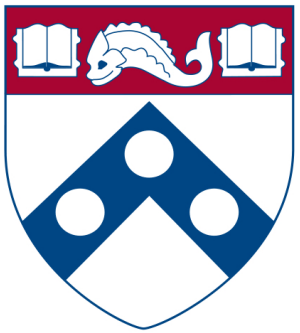


Penn Engineering

Online Learning

Video 4.1

Arvind Bhusnurmath

Some of the slides in this deck were reproduced with the permission of Dr. David Matuszek.

Topics

- Overloading
- How to have multiple methods with the same name

Overloading

- One component of polymorphism
- Polymorphism – ”the condition of existing in several forms”
- In this case it is a method existing in several forms in the same class.

Why is overloading needed?

Assume you are printing to the console. There are two ways of designing the print method

- Method 1 – printInt, printDouble, printString, ….
- Method 2 – just create one method called print. But Java insists that you specify the datatype. So that will not work ☹
- Method 3 – create a bunch of methods all of which are called print. Use the argument datatype to distinguish between them

Method 3 wins!
Overloading

Rules for overloading

- Signature of the method – the name of the method, the datatypes of the arguments .
  - includes the number of arguments
  - includes the order in which they occur

```
public String subString(String s1, int a, int b){
}
```

- The signature is subString(String, int, int)
- In Java a method signature does not include the return datatype

Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same
name, different parameters
        System.out.println("double d = " +
d);
    }
}
```

int i = 5; myPrint(i);
double d = 5.0; myPrint(d);

Why overload a method?

- So you can use the same names for methods that do essentially the same thing
- Example: println(int), println(double), println(boolean), println(String), etc.
- So you can supply defaults for the parameters:

```
int increment(int amount) {
    count = count + amount;
    return count;
}
int increment() {
    return increment(1);
}
```

- Notice that one method can call another of the same name

Why overload a method?

So you can supply additional information:

```
 void printResults() {
    System.out.println("total = " + total + ", average =
" + average);
 }
 void printResult(String message) {
    System.out.println(message + ": ");
    printResults();
 }
```

DRY (Don't Repeat Yourself)

When you overload a method with another, very similar method, only one of them should do most of the work:

```
void debug() {
    System.out.println("first = " + first );
     for (int i = first; i <= last; i++) {
        System.out.print(dictionary[i] + "  ");
     }
    System.out.println();
}

void debug(String s) {
    System.out.println("At checkpoint " + s + ":");
    debug();
}
```

Legal assignments

- Widening is legal (going to more general data type)
- Narrowing is illegal (unless you cast)
- All ints are doubles but all doubles are not ints, so Java gives you an error unless

```
class Test {
    public static void main(String args[]) {
        double d;
        int i;
        d = 5;                  // legal
        i = 3.5;                // illegal
        i = (int) 3.5;          // legal
    }
}
```

Legal method calls

- Legal because parameter transmission is equivalent to assignment
- myPrint(5) is like saying
  double d = 5;
  System.out.println(d);

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
    }

    static void myPrint(double d) {
        System.out.println(d);
    }
}
```

Illegal method calls

- Illegal because parameter transmission is equivalent to assignment
- myPrint(5.0) is like
  int i = 5.0;
  System.out.println(i);

```
class Test {
    public static void main(String args[]) {
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println(i);
    }
}
```

myPrint(int) in Test cannot be applied to (double)

Java uses the most specific method

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }
    static void myPrint(double d) {
        System.out.println("double: " + d);
    }
    static void myPrint(int i) {
        System.out.println("int: " + i);
    }
}
```

int:5
double: 5.0

Multiple constructors I

You can "overload" constructors as well as methods:

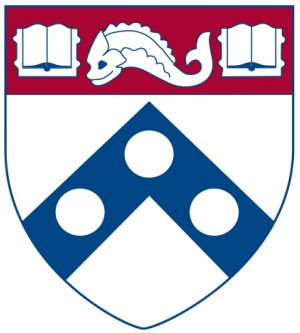
```
Counter() {
    count = 0;
}

Counter(int start) {
    count = start;
}
```

Multiple constructors 2

- One constructor can "call" another constructor in the same class, but there are special rules
- You call the other constructor with the keyword this
- The call must be the *very first thing* the constructor does

```
Point(int x, int y) {
    this.x = x;
    this.y = y;
    sum = x + y;
}
Point() {
    this(0, 0);
}
```



Penn Engineering

Online Learning

Video 4.2

Arvind Bhusnurmath

Some of the slides in this deck were reproduced with the permission of Dr. David Matuszek.

Topics

- Method Overriding

Extending a class (the “is a” relationship)

- Use the actual word ‘extends’
- class Square extends Rectangle
- class Goalkeeper extends Player
- You can only extend one class

Superclass Construction I

- The very first thing any constructor does, automatically, is call the *default* constructor for its superclass

```
class Foo extends Bar {
     Foo() { // constructor
         super(); // invisible call to superclass
 constructor
         ...
```

- You can replace this with a call to a *specific* superclass constructor
- Use the keyword super
- This must be the *very first thing* the constructor does

```
class Foo extends Bar {
     Foo(String name) { // constructor
         super(name, 5); // explicit call to superclass
 constructor
        ...
```

Superclass Construction 2

- Unless you specify otherwise, every constructor calls the *default* constructor for its superclass

```
class Foo extends Bar {
     Foo() { // constructor
         super(); // invisible call to superclass
 constructor
         ...
```

- You can use this(...) to call another constructor in the same class:

```
class Foo extends Bar {
     Foo(String message) { // constructor
         this(message, 0, 0); // your explicit call to
 another constructor
         ...
```

Superclass Construction 3

- You can use super(...) to call a specific *superclass* constructor

```
class Foo extends Bar {
     Foo(String name) { // constructor
         super(name, 5); // your explicit call to some
 superclass constructor
         ...
```

- Since the call to another constructor must be the very first thing you do in the constructor, you can only do one of the above

Overriding

```
class Animal {
    public static void main(String args[])
{
        Animal animal = new Animal();
        Dog dog = new Dog();
        animal.print();
        dog.print();
    }
    void print() {
        System.out.println("Superclass
Animal");
    }
}
public class Dog extends Animal {
    void print() {
        System.out.println("Subclass Dog");
    }
}
```

- This is called overriding a method
- Method print in Dog overrides method print in Animal

How to override a method

- Create a method in a subclass having the same *signature* as a method in a superclass
- That is, create a method in a subclass having the same name and the same number and types of parameters
- Parameter *names* don't matter, just their *types*
- Restrictions:
  - The return type must be the same
  - The overriding method cannot be *more private* than the method it overrides (ignore this bullet point for now)

Why override a method?

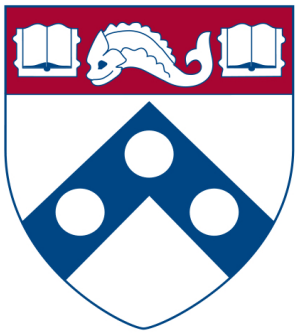
```
Dog dog = new Dog();
System.out.println(dog);
```

- Prints something like Dog@feda4c00
- The println method calls the toString method, which is defined in Java's top-level Object class
- Hence, every object *can* be printed (though it might not look pretty)
- Java's method public String toString() can be overridden

If you add to class Dog the following:

```
 public String toString() {
    return name;
}
```

Then System.out.println(dog); will print the dog's name, which may be something like: Fido



Penn Engineering

Online Learning

Video 4.3

Arvind Bhusnurmath

Topics

- Common examples of overriding
  - toString method
  - equals method

The Object class

- In Java, every class inherits from the Object class
- Think of the Object class as the most general class
- Every class that we define is lower in the hierarchy and becomes more and more specific

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true

toString()

- It is almost always a good idea to override `public String toString()` to return something "meaningful" about the object
- When debugging, it helps to be able to print objects
- When you print objects with `System.out.print` or `System.out.println`, they automatically call the objects `toString()` method
- When you concatenate an object with a string, the object's `toString()` method is automatically called

28

Calling toString() explicitly

- You can call `toString()` explicitly just like you would any other method
- Used in cases when you have to pass a string form of an object to another method.
- Can be used in unit testing to check if two objects are the same.
- For example you have 2 Person objects. You could decide to use
  `assertEquals(person1.toString(), person2.toString())`
- There are better ways to do this though.

Equality

- Consider these two assignments:

```
Thing thing1 = new Thing();
Thing thing2 = new Thing();
```

  Are these two "Things" equal?
  That's up to the programmer!

- But consider:

```
Thing thing3 = new Thing();
Thing thing4 = thing3;
```

  Are these two "Things" equal?
  Yes, because they are the ***same*** Thing!

The equals method

- Primitives can always be tested for equality with ==
- For objects, == tests whether the two are the ***same*** object
- Two strings "abc" and "abc" *may or may not be* == !
- Objects can be tested with the method

```
public boolean equals(Object o)
```

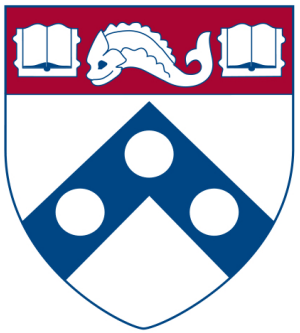
- Unless overridden, this method just uses ==
- It is overridden in the class String
- It is *not* overridden for arrays; == tests if its operands are the *same* array

Morals:

- Never use == to test *equality* of Strings or arrays or other objects
- Use equals for Strings, `java.util.Arrays.equals(a1, a2)` for arrays
- If you test your own objects for equality, override equals

The equals method in unit testing

- `assertEquals` in a Junit test uses the overriden(hopefully) method of the objects being compared.
- `assertArrayEquals` – when used on a array of objects the equals method is used for every index
- Consider array1 and array2 as arrays of Object
  `array1[i].equals(array2[i])`
  needs to be true for every index i



Penn Engineering

Online Learning

Video 4.4

Arvind Bhusnurmath

Some of the slides in this deck were reproduced with the permission of Dr. David Matuszek.

Topics

- Abstract classes

Abstract methods

- An abstract method is a method without any implementation

```
public abstract void draw(int size);
```

- Notice that the body of the method is completely missing. It is just the first line and then is terminated with a ;

Abstract class

- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword abstract

```
abstract class MyClass {...}
```

- An abstract class is incomplete
- It has "missing" method bodies
- You cannot instantiate (create a new instance of) an abstract class

Using an Abstract class

- Extend an abstract class before you can use it
- If the subclass defines all the inherited abstract methods, it is "complete" and can be instantiated.
- If the subclass does not define all the abstract methods then it too must be abstract.
- You can declare a class to be abstract even if it does not have any abstract methods.
  - This prevents the class from being instantiated.

Why have an abstract class

- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc.
- You don't want to allow creation of a "Shape"
  - Only particular shapes make sense, not generic ones
  - If Shape is abstract, you can't create a new Shape
  - You can create a new Oval, a new Rectangle, etc.
- Abstract classes are good for defining a general category containing specific, "concrete" classes

Example abstract class

```
public abstract class Animal {
    abstract int eat();
    abstract void breathe();
}
```

- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the eat() and breathe() methods

Potential Problem

```
class Shape { ... }
class Star extends Shape {
    void draw() { ... }
    ...
}
class Crescent extends Shape {
    void draw() { ... }
    ...
}
```

- `Shape someShape = new Star();`
  - This is legal, because a Star is a Shape
- `someShape.draw();`
  - This is a syntax error, because some Shape might not have a draw() method
  - Remember: A class knows its superclass, but not its subclasses

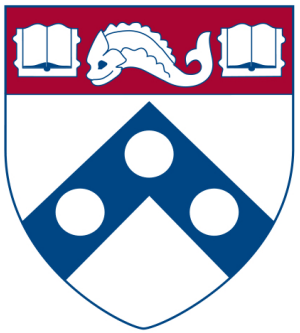
Usage of Abstract methods

- Suppose you are making a GUI, and you want to draw a number of different "shapes" (marbles, pegs, frogs, stars, etc.)
  - Each class (Marble, Peg, etc.) has a draw method
  - You make these subclasses of a class Shape, so that you can create an ArrayList<Shape> shapes to hold the various things to be drawn
  - You would like to do
    for (Shape s : shapes) s.draw();
  - This isn't legal!
- Every class "knows" its superclass, but a class doesn't "know" its subclasses
  - *You* may know that every subclass of Shape has a draw method, but Java doesn't
- Solution 1: Put a draw method in the Shape class
  - This method will be inherited by all subclasses, and will make Java happy
  - But what will it draw?
- Solution 2: Put an **abstract** draw method in the Shape class
  - This will also be inherited (and make Java happy), but you don't have to define it
  - You do, however, have to make the Shape class abstract
  - This way, Java knows that only "concrete" objects have a draw method

Solving the problem using abstract method

```
abstract class Shape {
    abstract void draw();
}
class Star extends Shape {
    void draw() { ... }
    ...
}
class Crescent extends Shape {
    void draw() { ... }
    ...
}
```

- `Shape someShape = new Star();`
  - This is legal, because a Star is a Shape
  - However, `Shape someShape = new Shape();` is no longer legal
- `someShape.draw();`
  - This is legal, because every actual instance must have a draw() method



Penn Engineering

Online Learning

Video 4.5

Arvind Bhusnurmath

Some of the slides in this deck were reproduced with the permission of Dr. David Matuszek.

Topics

- Interfaces

What is an Interface?

- "An interface is a group of related methods with empty bodies" – from the official Java documentation
- Most common way of specifying that a class follows a certain design.

The implements keyword

- Like signing a contract
- Agreeing to write certain methods.

Interfaces

- An interface declares (describes) methods but does not supply bodies for them

```
interface KeyListener {
   public void keyPressed(KeyEvent e);
   public void keyReleased(KeyEvent e);
   public void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly public and abstract
  - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
  - An interface is like a very abstract class—none of its methods are defined
- An interface may also contain constants (final variables)

When to write an interface

- You will frequently use the supplied Java interfaces
- Sometimes you will want to design your own
- You would write an interface if you want classes of various types to all have a certain set of capabilities
- For example, if you want to be able to create grocery items, you might define an interface as:

```
public interface Item{
    salePrice();
 }
```

implements != extends

- You extend a class, but you implement an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like

Example:

```
class MyListener
        implements KeyListener, ActionListener { … }
```

implements != signing a binding contract

- When you say a class implements an interface, you are promising to define all the methods that were declared in the interface

Example:

```
class MyKeyListener implements KeyListener {
      public void keyPressed(KeyEvent e) {...};
      public void keyReleased(KeyEvent e) {...};
      public void keyTyped(KeyEvent e) {...};
}
```

The "..." indicates actual code that you must supply

- Now you can create a new MyKeyListener

Do we have to write all the methods?

- It is possible for a class to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener {
    public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it *must* be an abstract class
- You must label it as such with the keyword abstract
- You can even extend an interface (to add methods):

```
interface FunkyKeyListener extends KeyListener { ...
}
```

Why interfaces?

Reason 1: A class can only extend one other class, but it can implement multiple interfaces

- This lets the class fill multiple "roles"
- In writing user interfaces it is common to have a class be able to handle different user interactions.

Example:

```
class MyApplet extends Applet
          implements ActionListener, KeyListener {
  ...
  }
```

Reason 2: You can write methods that work for more than one kind of class

Methods for more than one class

You can write methods that work with more than one class

```
interface RuleSet {
    boolean isLegal(Move m, Board b);
    void makeMove(Move m);
}
```

Methods for more than one class

```
class CheckersRules implements RuleSet { // one implementation
    public boolean isLegal(Move m, Board b) { ... }
    public void makeMove(Move m) { ... }
}

class ChessRules implements RuleSet { ... } // another
implementation
RuleSet rulesOfThisGame = new ChessRules();

if (rulesOfThisGame.isLegal(m, b)) {
      rulesOfThisGame.makeMove(m);
}
```

This statement is legal because, whatever kind of RuleSet object rulesOfThisGame is, it must have isLegal and makeMove methods

instanceof

- instanceof is a keyword that tells you whether a variable "is a" member of a class or interface

- For example, if

```
class Dog extends Animal implements Pet {...}
Animal fido = new Dog();
```

then the following are all true:

```
fido instanceof Dog
fido instanceof Animal
fido instanceof Pet
```

- instanceof is seldom used
  - When you find yourself wanting to use instanceof, think about whether the method you are writing should be moved to the individual subclasses