Polymorphism
signatures

- in any programming language, a signature is what distinguishes one function or method from another
- in C, every function has to have a different name
- in Java, two methods have to differ in their names, **or** in the number or types of their parameters
  - `foo(int i)` and `foo(int i, int j)` are different
  - `foo(int i)` and `foo(int k)` are the same
  - `foo(int i, double d)` and `foo(double d, int i)` are different
- in C++, the signature also includes the return type
  - but not in Java!
poly + morphism

• poly is Latin for “many”; morph is Latin for “form”

• in Java, a method or constructor may have multiple signatures
  • this is called overloading
  • EXAMPLE:
    Cat c1 = new Cat();
    Cat c2 = new Cat(“Felix”); // overloaded constructor
    c1.speak();
    c1.speak(3); // overloaded method

• overloading permits multiple ways to construct an object, or multiple ways to call a method
polymorphism in inheritance

• a method or constructor may be modified in a subclass during inheritance
  • this is called **overriding**
  • EXAMPLE:
      // assuming both Cat and Dog extend Animal, and Animal has
      // a speak() method defined:
      Cat c = new Cat();
      c.speak(); // overridden to speak specifically like a Cat
      Dog d = new Dog();
      d.speak(); // overridden to speak specifically like a Dog

• if you override a superclass **variable**, you may get unintended consequences
  • this is called **shadowing**
  • it is generally to be avoided
an example of 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
double d = 5.0
DRY (Don’t Repeat Yourself)

• when you overload a method, only one of the similar methods should do most of the work:

```java
void debug() {
    System.out.println("first = " + first + ", last = " + last);
    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();
}
```
another reason for polymorphism

- you may want to do “the same thing” with different kinds of data:
  - class Student extends Person {
    ...
    void printInformation() {
      printPersonalInformation();
      printGrades();
    }
  }
  - class Professor extends Person() {
    ...
    void printInformation() {
      printPersonalInformation();
      printResearchInterests();
    }
  }
- Java’s print and println methods are heavily overloaded
- if Person has a method called “printInformation”, it is **overridden**
- if Person has no such method, it’s still polymorphism—2 different implementations (in different classes) that bear the same name
legal assignments

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

• widening a number is legal
• narrowing a number is illegal (unless you cast)
legal method calls

class Test {
    public static void main(String args[]) {
        myPrint(5);
    }
    static void myPrint(double d) {
        System.out.println(d);
    }
}

5.0

• legal because parameter transmission is equivalent to assignment
• myPrint(5) is like double d = 5; System.out.println(d);
illegal method calls

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)

• illegal because parameter transmission is equivalent to assignment
• myPrint(5.0) is like int i = 5.0; System.out.println(i);
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 1

- You can “overload” constructors as well as methods:

  - Counter() {
      count = 0;
  }

  Counter(int start) {
      count = start;
  }

multiple constructors II

• 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);
  }
• a common reason for overloading constructors is (as above) to provide default values for missing parameters