Polymorphism
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!
Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- In Java, **polymorphism** refers to the fact that you can have multiple methods with the same name in the same class

There are two kinds of polymorphism:

- **Overloading**
  - Two or more methods with different signatures
- **Overriding**
  - Replacing an inherited method with another having the same signature
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
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:
  ```java
  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
- So you can supply additional information:
  ```java
  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:

```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();
}
```

```java
void debug(String s) {
    System.out.println("At checkpoint " + s + ":");
    debug();
}
```
Another reason to overload methods

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
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** is legal
- **Narrowing** is illegal (unless you cast)
Legal method calls

Legal because parameter transmission is equivalent to assignment

myPrint(5) is like double d = 5; System.out.println(d);
Illegal method calls

```java
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 I

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

```java
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

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

**Superclass construction I**
Superclass construction II

- 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
      ...
    }
  }

- 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
Shadowing

class Animal {
    String name = "Animal";
    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        System.out.println(animal.name + " " + dog.name);
    }
}

public class Dog extends Animal {
    String name = "Dog";
}

Animal Dog

- This is called shadowing—name in class Dog shadows name in class Animal
An aside: Namespaces

- In Python, if you named a variable `list`, you could no longer use the `list()` method.
- This sort of problem is very rare in Java.
- Java figures out what kind of thing a name refers to, and puts it in one of seven different namespaces:
  - package names
  - type names
  - field names
  - method names
  - local variable names (including parameters)
  - Labels
  - enums
- This is a separate issue from overloading, overriding, or shadowing.
Overriding

This is called **overriding** a method

Method `print` in **Dog** overrides method `print` in **Animal**

A subclass variable can *shadow* a superclass variable, but a subclass method can *override* a superclass method

```java
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");
    }
}
```

Superclass Animal
Subclass Dog
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.
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:
  ```java
  public String toString() {
    return name;
  }
  ```
  Then `System.out.println(dog);` will print the dog’s name, which may be something like: `Fido`
More about `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
  - You can explicitly call an object’s `toString()` method
    - This is sometimes helpful in writing unit tests; however...
    - Since `toString()` is used for printing, it’s something you want to be able to change easily (without breaking your test methods)
    - It’s usually better to write a separate method, similar to `toString()`, to use in your JUnit tests
Consider these two assignments:
```java
Thing thing1 = new Thing();
Thing thing2 = new Thing();
```
Are these two “Things” equal?
- That’s up to the programmer!

But consider:
```java
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)` in `java.lang`.
  - 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`
Calling an overridden method

- When your class overrides an inherited method, it basically “hides” the inherited method
- Within this class (but not from a different class), you can still call the overridden method, by prefixing the call with `super`.
  - Example: `super.printEverything();`
- You would most likely do this in order to observe the DRY principle
  - The superclass method will do most of the work, but you add to it or adjust its results
  - This isn’t a call to a constructor, and can occur anywhere in your class (it doesn’t have to be first)
Summary

- You should **overload** a method when you want to do essentially the same thing, but with different parameters.
- You should **override** an inherited method if you want to do something slightly different than in the superclass.
  - It’s almost always a good idea to override `public void toString()` -- it’s handy for debugging, and for many other reasons.
  - To test your own objects for equality, override `public void equals(Object o)`.
  - There are special methods (in `java.util.Arrays`) that you can use for testing array equality.
- You should never intentionally **shadow** a variable.
“Java is focused on being a powerful but simple language, easy to read, with a consistent clear meaning. It is more important that Java programs be easy to read than to write.”

--Graham Hamilton