Recap: Parameter Passing

Object as parameter:
- Copy of pointer made

```
object: Giraffe1
sayHowTall()
neckLength
```

Prim as parameter:
- Copy of value

```
main
foo
4
number
```

Recap: Static Fields/Methods

- Static fields belong to whole class
- Nonstatic fields belong to instantiated object
- Static methods can only use static fields
- Nonstatic methods can use either nonstatic or static fields

```
class: Giraffe
getGiraffeCount()
numGiraffes
object: Giraffe1
neckLength
sayHowTall()
```

Recap: Variable Types and Scope

- Static variables
  - Declared within class
  - Associated with class, not instance
- Instance variables
  - Declared within class
  - Associated with instance
  - Accessible throughout object, lifetime of object
- Local variables
  - Declared within method
  - Accessible throughout method, lifetime of method
- Parameters
  - Declared in parameter list of method
  - Accessible throughout method, lifetime of method

Recap: Variable Types

```
class: Giraffe
getGiraffeCount()
int numGiraffes
object: Giraffe1
int neckLength
sayHowTall()
yell(String message)
int volume
```

```
object: Giraffe2
int neckLength
sayHowTall()
yell(String message)
int volume
```
Here's a puzzler...
How does System.out.println() accept different data types as parameters?

```java
public class PrintlnTest {
    public static void main(String[] args) {
        int a = 7;
        double b = 3.14159;
        boolean c = false;
        String d = "woohoo!";
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

> java PrintlnTest

```
> 7
> 3.14159
false
"woohoo!"
```

In other words, why doesn't this blow up? Can you construct a method that will accept different data types?

Method overloading

Java allows us to create methods with the same name but different parameter lists. This is useful when you want to perform similar operations on different types of data as well as different numbers of parameters. This is called method overloading.

```java
public class AvgTest {
    public static void main(String[] args) {
        int a = 7;
        double b = 3.14159;
        boolean c = false;
        String d = "woohoo!";
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

```java
public static double avg(double a, double b, double c) {
    return ((a + b + c) / 3);
}
```

```java
public static double avg(int a, int b) {
    return ((a + b) / 2);
}
```

Method overloading - different types

```java
public class OverloadTest {
    public static void main(String[] args) {
        int a = 7;
        double b = 3.14159;
        boolean c = false;
        System.out.println("I am an integer.");
        System.out.println("Am I a boolean? Yes? No?");
        System.out.println("Aye, I'm a String and proud of it!");
    }
}
```

Here's a puzzler...
How does System.out.println() accept different data types as parameters?

```java
public class PrintlnTest {
    public static void main(String[] args) {
        int a = 7;
        double b = 3.14159;
        boolean c = false;
        String d = "woohoo!";
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

> java PrintlnTest

```
> 7
> 3.14159
false
"woohoo!"
```

Method overloading - param list length

```java
public class AvgTest {
    public static void main(String[] args) {
        System.out.println(avg(10, 30, 20));
        System.out.println(avg(30, 20));
    }
}
```

```java
public static double avg(double a, double b) {
    return ((a + b) / 2);
}
```

```java
public static double avg(double a, double b, double c) {
    return ((a + b + c) / 3);
}
```
Method overloading

When two or more methods have the same name, Java uses the number of parameters, the types of the parameters, and/or the order of the types of parameters to distinguish between the methods.

The method's name, type, and order of its parameters is called its signature. If you try to create two methods with the same signature, the compiler will let you know.

Method overloading public class AvgTest3{
   public static void main(String[] args) {
      ... defined in AvgTest3
   }
}

required: float

Method overloading public class AvgTest2{
   public static void main(String[] args) {
      ... logic
   }  
}

1 error found:
File: AvgTest2.java [line: 13] Error: avg(double,double) is already defined in AvgTest2

Method overloading When two or more methods have the same name, Java uses the number of parameters, the types of the parameters, and/or the order of the types of parameters to distinguish between the methods.

The method's name, type, and order of its parameters is called its signature. If you try to create two methods with the same signature, the compiler will let you know.

The return type is not part of the signature. That is, you can't have two overloaded methods whose signatures differ only by the return type. Why? There's no way for Java to know from the method invocation which method was intended to be used, and it's not going to choose one at random, is it?

Method overloading public class AvgTest2{
   public static void main(String[] args) {
      System.out.println(avg(30,20));
   }
}

public static double avg(double a, double b) {
   return ((a + b) / 2);                     // different logic
}

public static double avg(double a, double b) // same signature
   return ((a + b) / 2.0);                   // different logic
}

1 error found:
Files: AvgTest2.java [lines: 13]
Error: avg(double,Double) is already defined in AvgTest2

Method overloading public class AvgTest3{
   public static void main(String[] args) {
      System.out.println(avg(30,20));
   }
}

public static double avg(double a, double b) {
   return ((a + b) / 2);                     // different logic
}

public static double avg(double a, double b) // same signature
   return ((a + b) / 2.0);                   // different logic
}

2 errors found:
Files: AvgTest2.java [lines: 13]
Error: avg(double,Double) is already defined in AvgTest3
Files: AvgTest3.java [lines: 15]
Error: possible loss of precision
found : double
required: float

Constructor overloading

Can we overload constructor methods? Of course! Here's our favourite program, the CokeMachine...

public class CokeMachine2{
   private static int totalMachines = 0;
   private int numberOfCans;
   public CokeMachine2(){
      numberOfCans = 10;
      System.out.println("Adding another machine to your empire with "+ numberOfCans + " can of Coke");
      totalMachines++;
   }
}
Constructor overloading

```
public static int getTotalMachines() {
    return totalMachines;
}

public int getNumberOfCans() {
    return numberOfCans;
}

public void buyCoke() {
    if (numberOfCans > 0) {
        numberOfCans = numberOfCans - 1;
        System.out.println("Have a Coke");
        System.out.println("cans remaining");
    } else {
        System.out.println("Sold Out");
    }
}
```

Constructor overloading

```
public class CokeMachine2 {
  private static int totalMachines = 0;
  private int numberOfCans;

  public CokeMachine2() {
    numberOfCans = 10;
    System.out.println("Adding another machine to your empire with 10 cans of Coke");
    totalMachines++;
  }

  public CokeMachine2(int n) {
    numberOfCans = n;
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```

Another vending-related opportunity

Let's say that you've been inspired by CPSC 111 and decide to create commercial vending-machine simulation software. To make this work, you'll need to accommodate vending machines beyond those that sell only Coca-Cola products. For example, you may want to include...

Pizza machines...

```
public class SimCoke2 {
  public static void main(String[] args) {
    System.out.println("Coke machine simulator");
    CokeMachine2 cs = new CokeMachine2();
    CokeMachine2 engr = new CokeMachine2(237);
    CokeMachine2 chan = new CokeMachine2(42);
    CokeMachine2 library = new CokeMachine2(9000);
    cs.buyCoke();
    engr.buyCoke();
  }
}
```

> java SimCoke2
Coke machine simulator
Adding another machine to your empire with 10 cans of Coke
Adding another machine to your empire with 237 cans of Coke
Adding another machine to your empire with 42 cans of Coke
Adding another machine to your empire with 9000 cans of Coke
Have a Coke
9 cans remaining

23

Beer machines...

```
```

> java SimCoke2
Coke machine simulator
Adding another machine to your empire with 10 cans of Coke
Adding another machine to your empire with 237 cans of Coke
Adding another machine to your empire with 42 cans of Coke
Adding another machine to your empire with 9000 cans of Coke
Have a Coke
9 cans remaining
```
...and even French fry machines!

Another vending-related opportunity
Furthermore, while recognizing that a pizza machine is not the same as a beer machine is not the same as a Coke machine, you'll want to take advantage of the fact these two distinct types of vending machines have much in common. How can you do this? Here's one way...

Interfaces
Informally, we've used the word "interface" to refer to the set of public methods (for example, getters and setters) through which we interact with an object.

There's also a more formal use of the word interface in Java. A Java interface is a collection of constants and abstract methods.

We create an interface by using the reserved word interface in what would be the class header, if an interface were a class (which it's not...think of it more as the syntactic specification for a class).

We could use the reserved word abstract in the abstract method headers, but we don't have to because the methods in an interface must be abstract.
Interfaces

An interface is not a class. An interface only prescribes what methods some other class or classes must have. That is, an interface specifies the headings for methods that must be defined in any class that implements the interface, but it doesn’t say what happens inside the methods.

from Absolute Java by Walter Savitch.

Implementing an interface

A class implements an interface by providing method implementations for each of the abstract methods defined in the interface. A class that implements an interface uses the reserved word implements followed by the interface name in the class header.
Implementing an interface

For a class to implement an interface, it must provide a definition for all methods in the interface. If there are unimplemented methods, the class must be declared to be an abstract class, but that's a topic for another time.

What do these relationships look like?

Generic Vending Machine
Beer Machine
French Fry Machine
Coke Machine
Pizza Machine

Why this stuff is very very cool

Because an object of a class that implements an interface is also an object of that interface type. That concept is the basis of an important object-oriented programming principle called polymorphism.

Polymorphism is derived from the word fragment poly and the word morpho in Greek, and it literally means "multiple forms".

*assuming you come from the planet Nerdtron

Why this stuff is very very cool

Polymorphism simplifies the processing of various objects in the same class hierarchy by using the same method call for any object in the hierarchy. We make the method call using an object reference of the interface. At run time, the Java Virtual Machine determines which class in the hierarchy the object actually belongs to and invokes the version of the method implemented for that class.

from Java 5 Illuminated by Anderson and Franceschi

Why this stuff is very very cool

public class SimCoke2005
{
    public static void main (String[] args) {
        VendingMachine foo1 = new CokeMachine2005();
        VendingMachine foo2 = new FrenchFryMachine2005();
        foo1.vendItem();
        foo2.vendItem();
    }
}

The little foos may look like VendingMachine objects to you and me, but Java knows the difference and finds the appropriate method for each foo. That makes our programming job a lot easier to do. Why?
Why this stuff is very very cool

Because the alternative is to write lots of chunks of code that look like sort of like this (if they were written in English):

```java
if we want to vend an item from foo1 and foo1 is a CokeMachine2005
then print "have a Coke"
else if we want to vend an item from foo1 and foo1 is a FrenchFryMachine2005
then print "have a cup of french fries"
else if we want to vend an item from foo1 and foo1 is a PizzaMachine2005
then ...
```

As the number of classes within the same hierarchy grows, so does the size of the chunks of code represented above. Eeyow!

---

A simple bunny example

```java
public interface Bunnies
{
    public void moveBunny(int direction);
}
```

A simple bunny example

```java
public class BigBunny implements Bunnies
{
    private int x, y;
    private int carrots;
    public BigBunny()
    {
        x = 5;
        y = 5;
        carrots = 10;
    }
    public void moveBunny(int direction)
    {
        if (direction == 12)
        {  
            y = y + 3;
            carrots = carrots - 2;
        }  
        else if (direction == 3)
        {  
            x = x + 3;
            carrots = carrots - 2;
        }  
        else if (direction == 6)
        {  
            y = y - 3;
            carrots = carrots - 2;
        }  
        else if (direction == 9)
        {  
            x = x - 3;
            carrots = carrots - 2;
        }  
        else
        {  
            System.out.println("Invalid direction");
        }
    }
}
```

A simple bunny example

```java
public class LittleBunny implements Bunnies
{
    private int x, y;
    private int carrots;
    public LittleBunny()
    {
        x = 5;
        y = 5;
        carrots = 10;
    }
    public void moveBunny(int direction)
    {
        if (direction == 12)
        {  
            y = y + 1;
            carrots = carrots - 1;
        }  
        else if (direction == 3)
        {  
            x = x + 1;
            carrots = carrots - 1;
        }  
        else if (direction == 6)
        {  
            y = y - 1;
            carrots = carrots - 1;
        }  
        else if (direction == 9)
        {  
            x = x - 1;
            carrots = carrots - 1;
        }  
        else
        {  
            System.out.println("Invalid direction");
        }
    }
}
```
**Interface caution**

Can't construct interface

Can only construct objects that belong to some class