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

```java
class PrintlnTest{
  public static void main(String[] args){
    System.out.println("Hello, World!");
    System.out.println(30);  // same signature
    System.out.println(30.30);  // same signature
    System.out.println(30.30,30);  // different logic
    System.out.println("Hello, World!");
  }
}
```

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

---

Recap: Favorite Colors

- record everybody’s favorite color
- how can we do “averages” per row?
- find the max
- keep array of vote counts for each color, for each row

---

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 AvgTest2{
  public static void main(String[] args){
    double a = 7;
    double b = 8;
    double c = 9;
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```

---

Constructor overloading

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

```java
public class CokeMachine{
  public CokeMachine(){
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```

---

Method overloading - different types

- public static double avg(double a, double b){
- public static double avg(double a, double b, double c){

---

Method overloading - param list length

- public static double avg(double a, double b){
- public static double avg(double a, double b, double c){

---

Method overloading - different types

- public static double avg(double a, double b){
- public static double avg(double a, double b, double c){

---

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.

```java
public class CokeMachine{
  public CokeMachine(){
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```

---

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.

```java
public class CokeMachine{
  public CokeMachine(){
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```

---

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.

```java
public class CokeMachine{
  public CokeMachine(){
    System.out.println("Adding another machine to your empire with " + numberOfCans + " cans of Coke");
    totalMachines++;
  }
}
```
Construct overloading
public static int getTotalMachines()  {    return totalMachines;  }

Constructor overloading
public class CokeMachine2005 implements VendingMachine{  private int itemsRemaining;  private int itemsSold;  public CokeMachine2005()  {    itemsRemaining = 10;    System.out.println("Adding another machine to your empire with ", 10 + " cans of Coke");  }  public CokeMachine2005(int n)  {    System.out.println("Adding another machine to your empire with ", n + " cans of Coke");    itemsRemaining = n;  }  public int getItemsRemaining()  {    return itemsRemaining;  }  public int getItemsSold()  {    return itemsSold;  }

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.

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...

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.

Implementing an interface
public class CokeMachine2005 implements VendingMachine{  private int itemsRemaining;  private int itemsSold;  public CokeMachine2005()  {    itemsRemaining = 10;    System.out.println("Adding another machine to your empire with ", 10 + " cans of Coke");  }  public CokeMachine2005(int n)  {    System.out.println("Adding another machine to your empire with ", n + " cans of Coke");    itemsRemaining = n;  }  public int getItemsRemaining()  {    return itemsRemaining;  }  public int getItemsSold()  {    return itemsSold;  }}
Implementing an interface

```java
public class FrenchFryMachine2005 implements VendingMachine {
  private int itemsRemaining;
  // other methods...

  public int getItemsRemaining() {
    return itemsRemaining;
  }

  public int getItemsSold() {
    return itemsSold;
  }
```

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.

```java
public class SimCoke2005 {
  public static void main (String[] args) {
    CokeMachine2005 foo1 = new CokeMachine2005();
    // call vending method using foo1 as a CokeMachine2005
    CokeMachine2005 foo2 = new FrenchFryMachine2005();
    // call vending method using foo2 as a FrenchFryMachine
  }
```

Interface caution

Can't construct interface

Can only construct objects that belong to some class.

```java
public class SimCoke2005 {
  public static void main (String[] args) {
    CokeMachine2005 foo1 = new CokeMachine2005();
    // can only construct CokeMachine2005 objects
    FrenchFryMachine2005 foo2 = new FrenchFryMachine2005();
    // can only construct FrenchFryMachine objects
  }
```

What do these relationships look like?

```
   Vending Machine
    |     |
    |     |
   Beer Machine
    |     |
   French Fry Machine
    |     |
   Cake Machine
    |     |
   Pizza Machine
```

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?