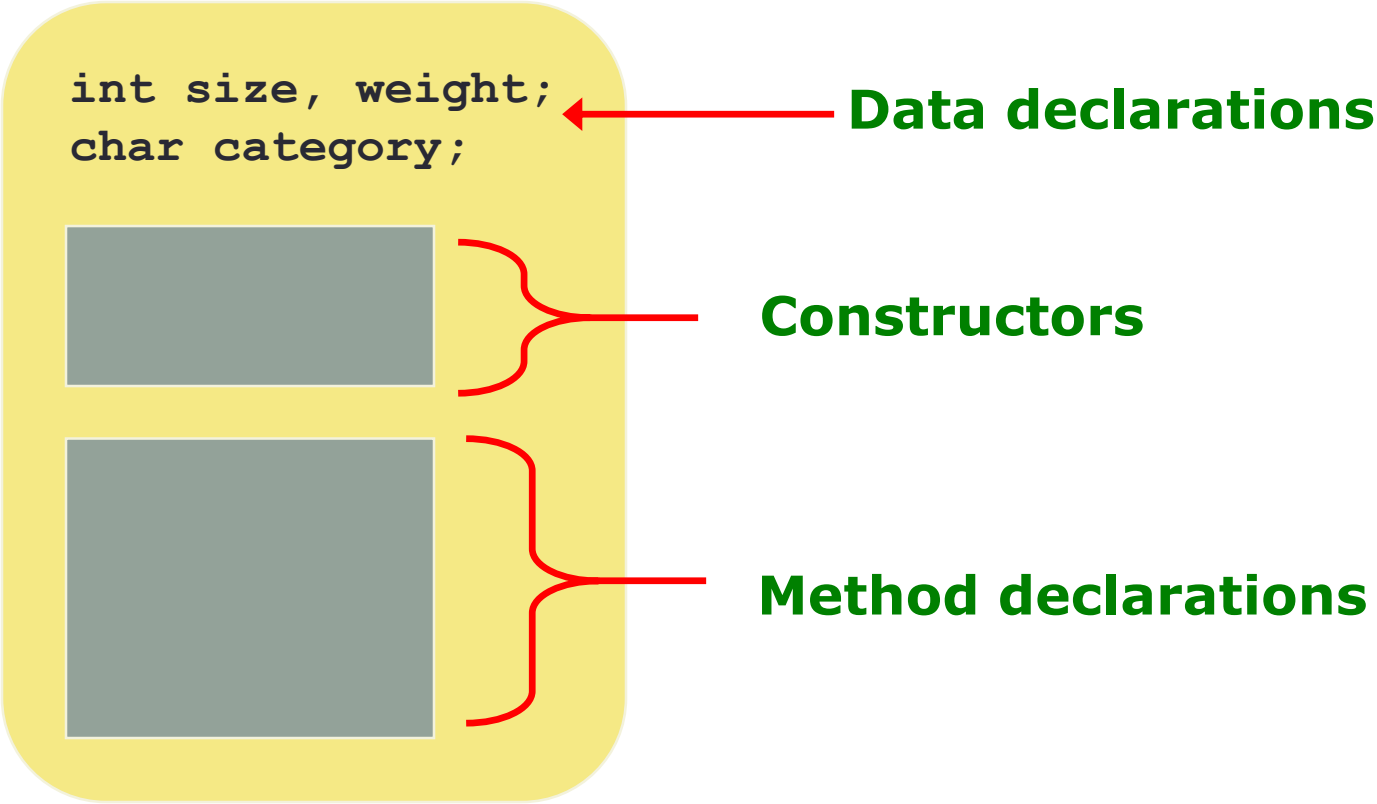
# CONSTRUCTORS

# Classes

- A class can contain data declarations, constructors and method declarations



```
public class Die {
 //instance data
   private final int MAX = 6;   // maximum face value
   private int faceValue;   // current value showing on the die
  //constructors
   Die(){
      this(1);
   }
   Die(int d) {
      faceValue = d;
   }
 //methods
   public int roll(){
      faceValue = (int)(Math.random() * MAX) + 1;
      return faceValue;
   }
   public void setFaceValue (int value) {
      faceValue = value;
   }
   public int getFaceValue(){
      return faceValue;
   }
   public String toString(){
      String result = Integer.toString(faceValue);
      return result;
   }
}
```

```
public class RollingDice{
  public static void main (String[] args){
    Die die1, die2;
    int sum;

    die1 = new Die();
    die2 = new Die();

    die1.roll();
    die2.roll();
    System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

    die1.roll();
    die2.setFaceValue(4);
    System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

    sum = die1.getFaceValue() + die2.getFaceValue();
    System.out.println ("Sum: " + sum);

    sum = die1.roll() + die2.roll();
    System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
    System.out.println ("New sum: " + sum);
  }
}
```

# Constructors

- A constructor with no parameters is referred to as a *default constructor*.
- Constructors must have the same name as the class itself.
- Constructors do not have a return type not even void.
- Constructors are invoked using the **new** operator when an object is created.
- Constructors play the role of initializing objects.

# Constructors Vs. Methods

- Like methods, **constructors can have any of the access modifiers**: public, protected, private, or default and can be overloaded
- Methods can have any valid return type, or no return type, in which case the return type is given as void. **Constructors have no return type, not even void.**
- **Constructors have the same name as their class**; by convention, methods use names other than the class name
- We **can not invoke constructors directly** like other methods, the constructors will be invoked when uses **new** operator

# Constructors

- A common error is to put a return type on a constructor, which makes it a **"regular"** method that happens to have the same name as the class

```
public void MyClass () {} // not a constructor
public MyClass() {} // constructor
```

- If the programmer does not have to define a constructor for a class, each class has a ***default constructor*** that accepts no parameters

# Overloaded Constructors

Constructors are a special kind of methods that are invoked to construct objects.

```
Die(int r) {
  faceValue = r;
}

Die(){
  faceValue = 1;
}

Die myDie = new Die(1);
```

# this Constructor

- Inside a constructor, you can use ***this*** to invoke another constructor in the same class.

- This is called ***explicit constructor invocation***.

- this constructor must be the **first statement and used only once within** the constructor body.

# this Constructor Example

```
Die(int r) {
  faceValue = r;
}

Die(){
  this(1);
}
```

# this REFERENCES

# this References

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method including constructor, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

**In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`**

# this References

- Inside methods
  - If parameter has same name as instance variable
    - Instance variable hidden
  - Use this.variableName to explicitly refer to the instance variable
  - Use variableName to refer to the parameter
  - Can help clarify program

# this Reference Example

```
class Student {
  private int id ;
  private String name ;

  public Student() {
         this(99999999, “unknown”) ;
  }

  public Student(int id) {
         this.id = id ;
  }


  public Student(int id, String name) {
         this.id = id ;
         this.name = name ;
  }

  public void setName(String name) {
         this.name = name ;
  }

}
```

# PASSING PARAMETERS

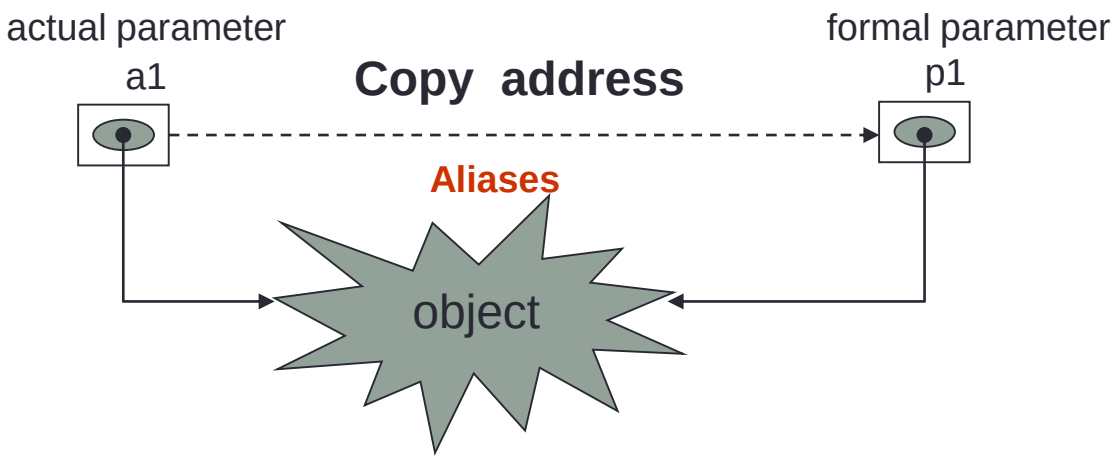
# Objects as Parameters

- Parameters in a Java method are *passed by value*
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

# Passing Parameters

- **Primitive Variables**


- **Reference Variables**



# Passing Objects to Methods

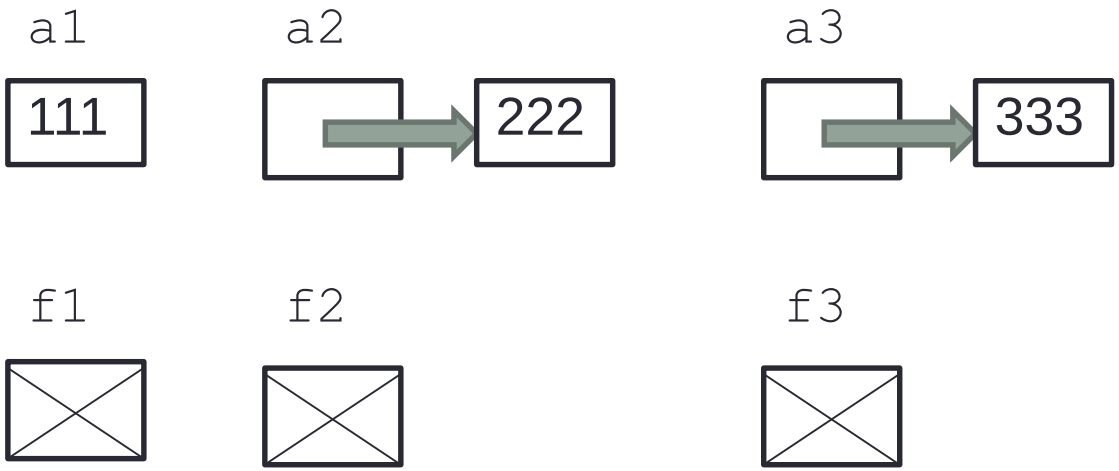
- What a method does with a parameter may or may not have a permanent effect (outside the method)
  - See ParameterTester.java
  - See ParameterModifier.java
  - See Num.java
- Note the difference between changing the internal state of an object versus changing which object a reference points to

```
public class Num{
   private int value;

   public Num (int update){
      value = update;
   }

   public void setValue (int update){
      value = update;
   }

    public String toString (){
          return value + "";
    }
}
```

```
public class ParameterModifier{
   public void changeValues (int f1, Num f2, Num f3){
      System.out.println ("Before changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num (777);

      System.out.println ("After changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```

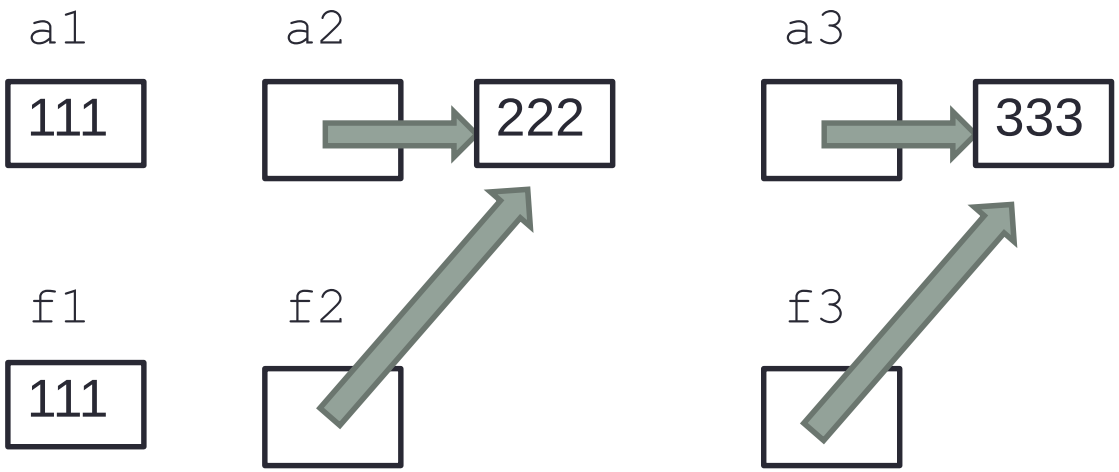
```
public class ParameterTester{
  public static void main (String[] args)  {
      ParameterModifier modifier = new ParameterModifier();

      int a1 = 111;
      Num a2 = new Num (222);
      Num a3 = new Num (333);

      System.out.println ("Before calling changeValues:");
      System.out.println ("a1\ta2\ta3");
      System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

      modifier.changeValues (a1, a2, a3);

      System.out.println ("After calling changeValues:");
      System.out.println ("a1\ta2\ta3");
      System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
   }
}
```

# STEP 1 – Before invoking changeValues




= Undefined

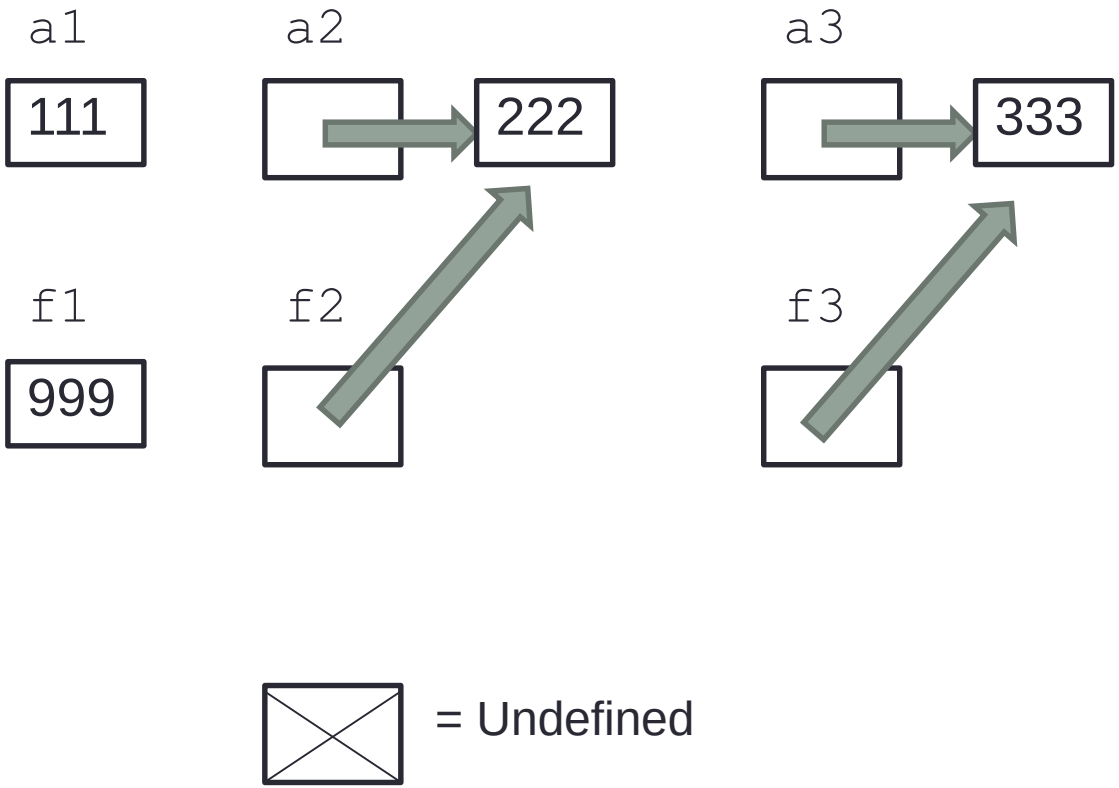
# STEP 2
# modifier.changeValues(a1, a2, a3);



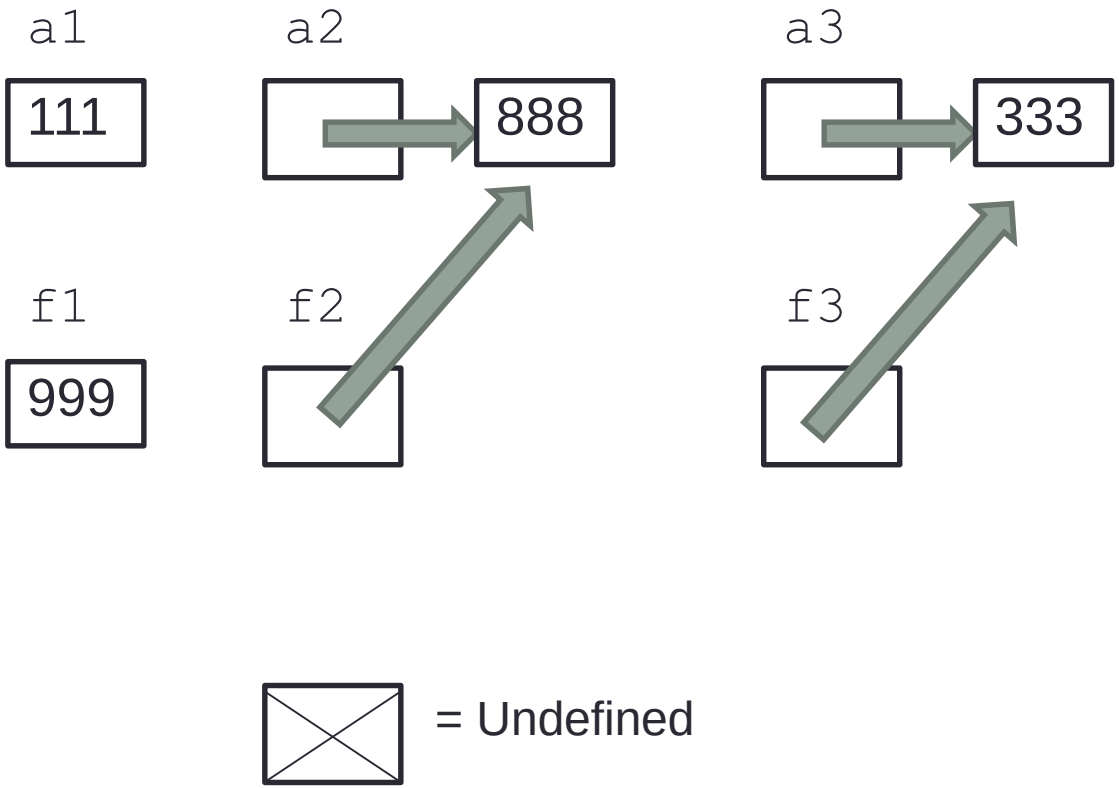


= Undefined

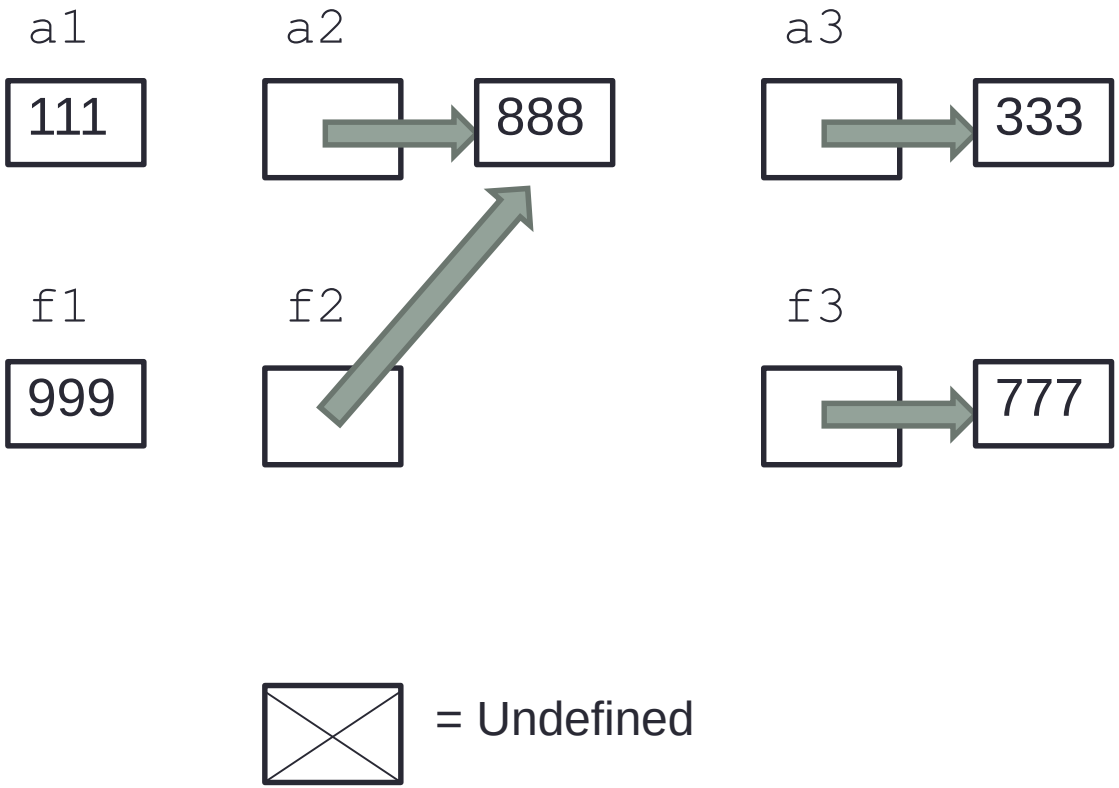
# STEP 3 – f1=999;



# STEP 4 – f2.setValue(888);



# STEP 5 – f3=new Num(777);



# STEP 6
# After returning from changeValues