MIT Accelerating Information Technology Innovation

Lecture 3: Classes and Objects; Encapsulation and References; Static Fields and Methods

http://aiti.mit.edu

MIT Accelerating Information Technology Innovation

Classes and Objects

What do we know so far?

- Primitives: int, float, double, boolean, char
- Variables: Stores values of one type.
- Arrays: Store many of the same type.
- Control Structures: If-then, For Loops.
- Methods: Block of code that we can pass arguments to and run multiple times.
- Is this all we want?

Object-Oriented Programming

- Programming using *objects*

- An object represents an entity
  - Real world object: String, car, watch, …
  - Abstract object: list, network connection, …

- Objects have two parts:
  - State: Properties of an object.
  - Behavior: Things the object can do.

Objects

- Car Example:
  - State: Color, engine size, automatic
  - Behavior: Brake, accelerate, shift gear
- Person Example:
  - State: Height, weight, gender, age
  - Behavior: Eat, sleep, exercise, study

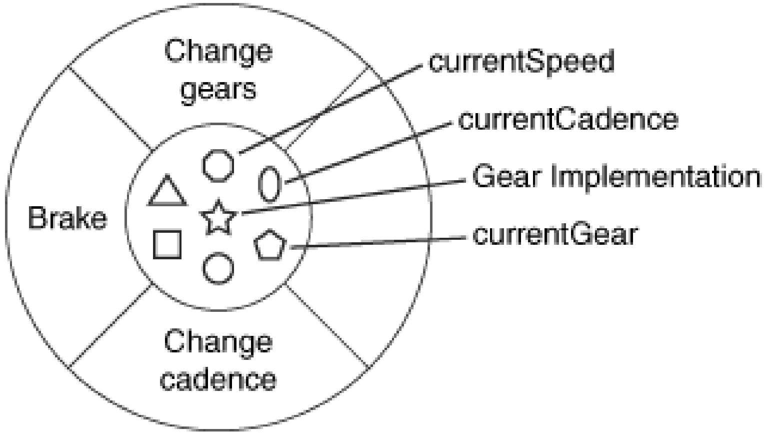
Why use objects?

- Modularity: Once we define an object, we can reuse it for other applications.
- Abstraction: Programmers don’t need to know exactly how the object works. Just the interface.
- Encapsulation: Hide the internal mechanisms to keep consistency.

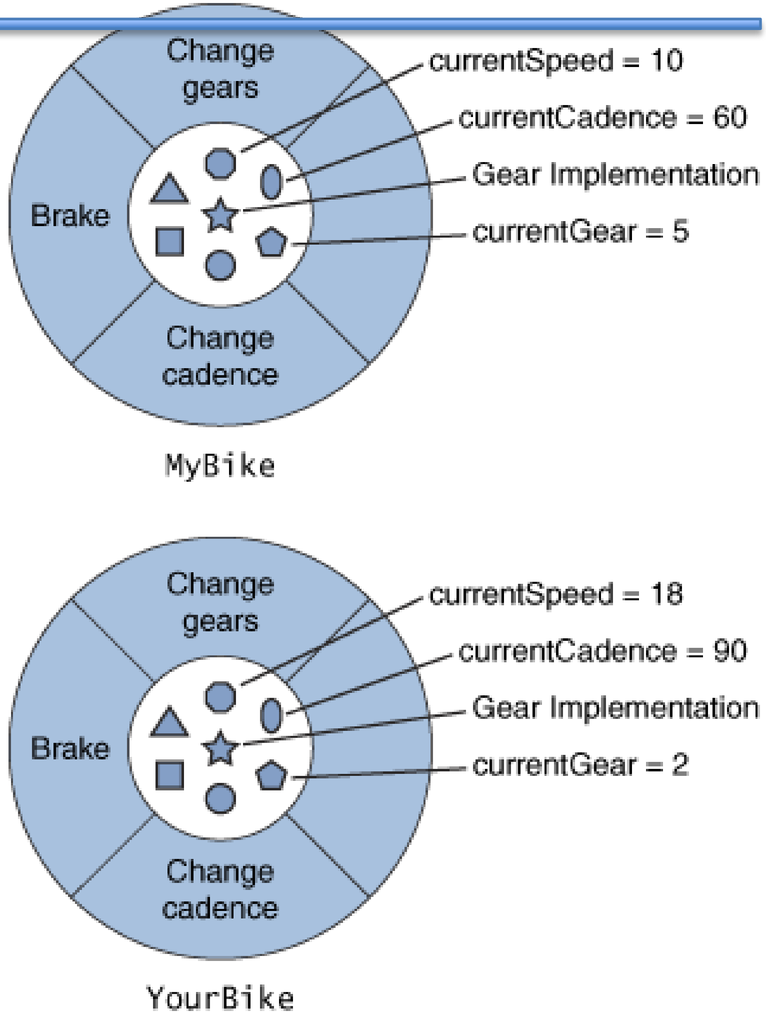
Abstraction

- We abstract away details to deal with complex problems.
  - Necessary for forming relationships between complex pieces of code.
  - The art is knowing which details to hide away and which to preserve.
  - What is a forms of abstraction have we seen so far?
- Example:
  - Different cars can use the same parts.
  - You don’t need to know how an engine works in order to drive a car.

Classes



A Bicycle Class



Two instances of the Bicycle Class

Our first Class: LightSwitch

```
class LightSwitch {
 boolean isOn = true;
}
```

- What is the state of a LightSwitch?
- State stored in fields; here it’s “isOn”.
- Fields are accessed using:
  - variableName.fieldName
  - (We’ll discuss other types of fields later)
- What are the behaviors of a LightSwitch?

Our First Class: LightSwitch

```java
class LightSwitch {

}
```

- `class` keyword tells Java you are creating a class
- The class must reside in a file named *ClassName*.java
  - Ex: LightSwitch.java
- Currently, our class does nothing…

Adding State

```
class LightSwitch {
 boolean isOn = true;
}
```

- What is the state of a LightSwitch?
- State stored in fields; here it’s “isOn”.
- Fields are accessed using:
  - variableName.fieldName
  - (We’ll discuss other types of fields later)
- What are the behaviors of a LightSwitch?

Adding Behavior

```
class LightSwitch {
 boolean isOn = true;
 void flip() {
    this.isOn = !this.isOn;
 }
}
```

- We define methods in a class to add behavior
  - Methods change the state of the object and affect system state
- `this.isOn` accesses the `isOn` field.
- What behavior does LightSwitch have now?

this Keyword

- Reference to the current object
  - The object whose method is being called

- Used to access fields:

```
class SimpleClass {
   int x = 0;   //Field of SimpleClass

   void foo(int x) {
            this.x = x;
      }
}
```

Using Objects

```
public static void main(String[] args) {
     LightSwitch s = new LightSwitch();
     System.out.println(s.isOn);
     s.flip();
     System.out.println(s.isOn);
}
```

- The new keyword creates a new object.
- new must be followed by a constructor.
- We call methods like:
  - variableName.methodName(arguments)
- What does this code output?

Constructors

- Constructors initialize the object after memory is allocated.
  - We can pass constructors data needed during initialization

- Objects have a default constructor that takes no arguments, like LightSwitch()

Constructors

- We can define our own constructors that take any number of arguments.
  - LightSwitch(boolean startState)

- Constructors have NO return type and must be named the same as the class:
  - ClassName(argument signature) { body }

Constructors

```
class LightSwitch {
  boolean isOn;
  void flip() {
      this.isOn = !this.isOn;
  }
  LightSwitch(boolean startState) {
      this.isOn = startState;
  }
}
```

- The LightSwitch() constructor no longer works. How do we instantiate an object?

Multiple Constructors

- We can have multiple constructors.
- Constructors can call each other.

```
LightSwitch() {
   this(true);
}
LightSwitch(boolean startState){
   this.isOn = startState;
}
```

Review

- What two properties do objects have?
- What is the difference between a class and an object?
- What is a field?
- What does the this keyword mean?
- What does the new keyword do?
- What is a constructor?

BankAccount Example

```
public class BankAccount {
   double balance;
   String name;
   BankAccount(String name,
               double openBalance){
     this.name = name;
     this.balance = openBalance;
   } // Continued next slide
   …
```

BankAccount Example

```
…
double deposit(double amount) {
   balance += amount;
   return balance;
}
boolean withdraw(double amount) {
   if (amount < balance) {
      balance -= amount;
      return true;
   } else return false;
 }
}// End BankAccount Class
```

MIT Accelerating Information Technology Innovation

Object Encapsulation and References

Data Field Encapsulation

- Sometimes we want variables to be accessible only within the class itself
  - Hide from other classes

- Prevents undesired/incorrect tampering with variables by methods outside of the class
  - Maintain consistency of state

Without Encapsulation..

```
class BankAccount {
   //Fields
   double balance;
   String name;

   //constructor
   BankAccount(String name, double openBalance){
     this.name = name;
     this.balance = openBalance;
   }
}
```

In Another Class

```
class AnotherClass {
  static void main(String[] args) {
    //create bank account
    BankAccount mikesAccount =
        new BankAccount (“Mike”, 10000000);

    //some tampering…
    mikesAccount.name = “Zach”;
  }
}
```

This is not good for poor Mike!

Visibility Modifiers

- `public` – makes methods and data fields accessible by any other class
- `private` – makes methods and data fields accessible only from within its own class
- (neither) – similar to public but a bit more restricted

Example, BankAccount

```
class BankAccount {

   //data fields
   private double balance;
   private String name;

   //constructor
   BankAccount(String name, double openBalance){
     this.name = name;
     this.balance = openBalance;
   }
}
```

Common Object Oriented Practices

- **Accessors – *get*** the value of a data field
  - Sometimes called **getters**

- **Mutators** – **set** the value of a data field
  - Sometimes called **setters**

BankAccount, add accessors

```
public class BankAccount {
                                     –
                                     –
                                     –

    //accessors
    public double getBalance(){
      return balance;
    }

    public String getName(){
      return name;
    }
```

BankAccount, add mutators

```
//mutators
public void deposit(double amount){
   …
}

public void withdraw (double amount){
   …
}
```

Notice there is no access to the name data field! Now Zach can’t steal Mike’s account.

Now we are safe!

```
class AnotherClass {
  static void main(String[] args) {
    //create bank account
    BankAccount mikesAccount =
        new BankAccount (“Mike”, 5);

    //Illega
    mikesA        name    “Zach”;
    //Ille     .
    mikesA   unt.         = 100000000;
  }
}
```

private Methods

- Methods of a class that are declared private can only be called within the class.

```
private void setName(String newName) {
 …
}
```

Now we are safe!

```
class AnotherClass {
  static void main(String[] args) {
    //create bank account
    BankAccount mikesAccount =
        new BankAccount (“Mike”, 5);

    //Illegal, private method of Bank Account
    mikesAccount.setName(“Zach”);
  }
}
```

Accessibility Intuition

- Accessibility modifiers are not used for safety
  - There are ways around them in Java!
- They are used for encapsulation!
  - Hide unnecessary state/methods from user of class
  - Prevent access to state to maintain object consistency

Consistency Example

```
class Family {
    Person[] males;
    Person[] females;

    //want totalMembers = males + females
    int totalMembers = 0;
    …
    public void addFemale(Person person)…
    public void addMale(Person person)…
}
```

Inconsistent

```
class AnotherClass {
   void method() {
    Family myFam = new Family();
    myFam.addMale(new Person(“Mike”));
    myFam.addFemale(new Person(“Mary”));
    myFam.totalMembers = 10;
    //now myFam is inconsistent!
   }
}
```

A Better Way!

```
class Family {
    private Person[] males;
    private Person[] females;
    //want totalMembers = males + females
    private int totalMembers = 0;
    …
    public void addFemale(Person person) {
     females[…] = person;
     totalMembers++;
    }
}
```

Object References

- An object variable is really a reference to the object.
  - A pointer is a good way of thinking about it

- You must “dereference” the variable to access method and fields
  - Ex: `person.getName(), course.number`

References

- You can have 2 variables reference the same object

```
Integer a = new Integer(5);
Integer b = a;
//a and b reference the same object
```

Primitive Argument Passing

- Remember that primitive arguments are passed by value.

- If you change a primitive argument inside of a method, the variable in the calling method will remain unchanged.

Review:
Primitive Argument Passing

```
public static int meth(int a, int b) {

    a = a * 2;
    b = b * 3;
    return a + b;
}


public static void main(String[] args) {
    int x = 5;
    int y = 10;
    int z = 0;

    z = meth(x, y);
    //what is the value of x and y?
}
```

Object Argument Passing

- Object Arguments are pass by reference
  - **A copy is not made**

- Any changes to the object in the method are visible in the calling method

Object Argument Passing

```
void changeName(Person person) {

    person.setName(“Mike”);
}




public static void main(String[] args) {
    Person cory = new Person(“Cory”);

    changeName(person);

    //what is the value cory.getName()?
}
```

Static Fields and Methods

What You Know So Far

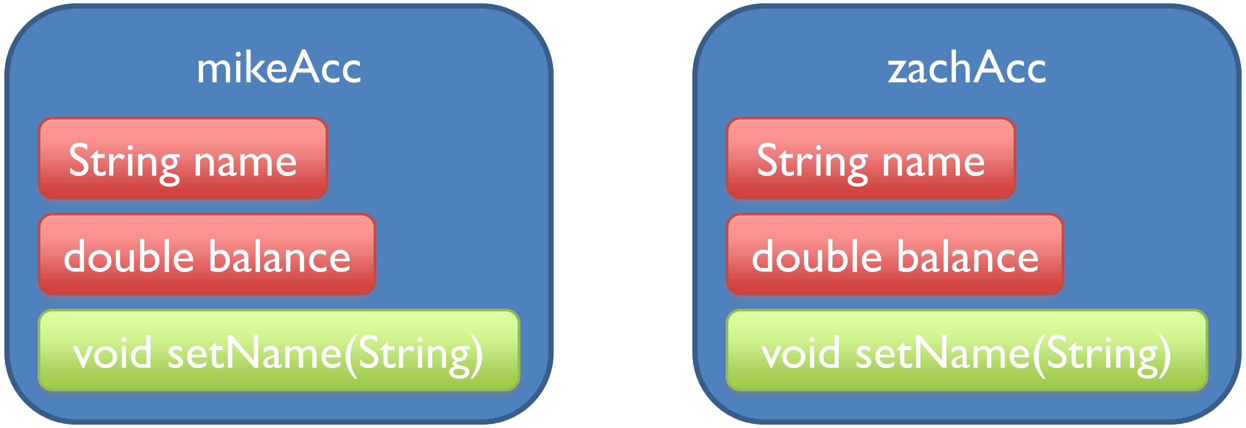
- Each object has its own copy of methods and fields:

```
class BankAccount {
    private String name;
    private double balance;
    public void withdraw(double amount) …
}
```

```
BankAccount mikeAcc = new BankAccount(“Mike”, 100);
BankAccount zachAcc = new BankAccount(“Zach”, 20);
```

Instance Fields and Methods

- Each object has its own copy of methods and fields:



Instance Fields and Methods

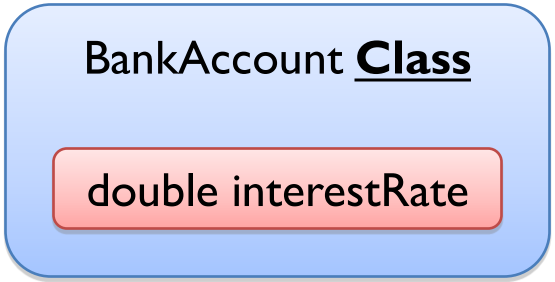
```
BankAccount mikeAcc = new BankAccount(“Mike”, 100);
BankAccount zachAcc = new BankAccount(“Zach”, 20);

System.out.println(mikeAcc.getBalance()); //100
System.out.println(zachAcc.getBalance()); //20

zachAcc.withdraw(19);

System.out.println(mikeAcc.getBalance()); //100
System.out.println(zachAcc.getBalance()); //1
```

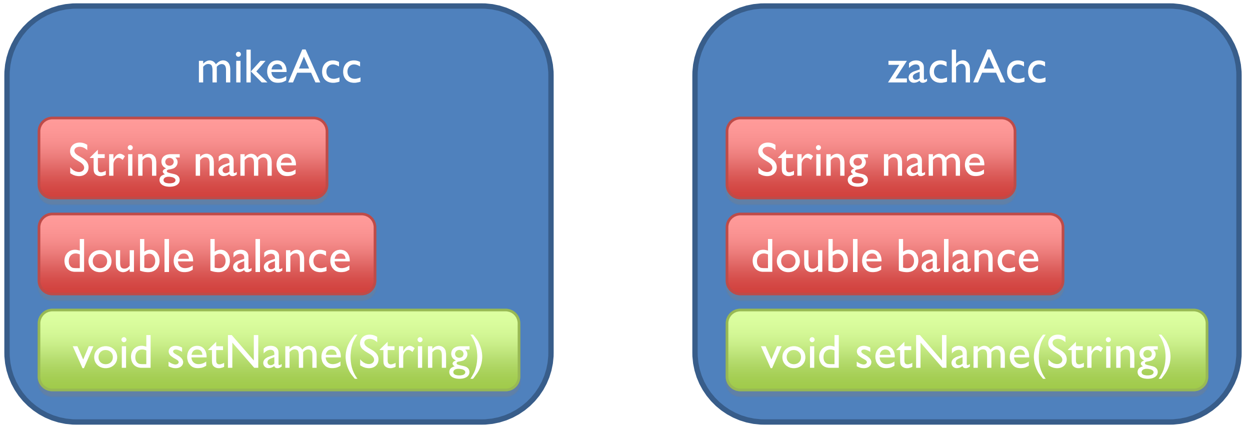
Shared Fields



BankAccount **Class**

double interestRate

- What if we wanted to make a field shared among all objects of a class?



mikeAcc

String name

double balance

void setName(String)

zachAcc

String name

double balance

void setName(String)

Static Fields

- A given class will only have one copy of each of its static fields
  - This will be shared among all the objects.

- Each static field exists even if **no** objects of the class have been created.

- Use the word **static** to declare a static field.

Static Fields

- Only one instance of a static field data for the entire class, not one per instance.

- "static" is a historic keyword from C/C++

Static Fields Example

```
class BankAccount {
    public static double interestRate = 0.02;
}

BankAccount mikeAcc = new BankAccount(“Mike”, 100);
BankAccount zachAcc = new BankAccount(“Zach”, 20);

System.out.println(mikeAcc.interestRate); //0.02
System.out.println(BankAccount.interestRate); //0.02

mikeAcc.interestRate = 0.05;
System.out.println(zachAcc.interestRate); //0.05
```

Counting Objects Created

```
public class BankAccount {
    private static int numAccounts = 0;

    public BankAccount(String name,
                                double balance) {
      numAccounts++;
      …
    }
}
```

Unique ID for Objects

```
public class BankAccount {
    private static int nextAccountNum = 0;
    private int accountNum;

    public BankAccount(String name,
                                  double balance) {
      accountNum = nextAccountNum++;
      …
    }
}
```

Array of All Objects Created

```
public class BankAccount {
   private static BankAccount[] accounts =
         new BankAccount[100];
   private static int nextAccountNum = 0;

   public BankAccount(String name,
                                    double balance) {
    accounts[nextAccountNum++] = this;
    …
   }
}
```

What would happen if we deleted this static modifier?

Array of All Objects Created

```
public class BankAccount {
    private BankAccount[] accounts =
          new BankAccount[100];
    private static int nextAccountNum = 0;

    public BankAccount(String name,
                                    double balance) {
     accounts[nextAccountNum++] = this;
     …
    }
}
```

More Static Field Examples

Constants used by a class:

- Usually used with final keyword
- Only need to have one per class; don’t need one in each object:

```
public static final double TEMP_CONVERT = 1.8;
```

- If variable TEMP_CONVERT is in class Temperature, it is invoked by:

```
double t = Temperature.TEMP_CONVERT * temp;
```

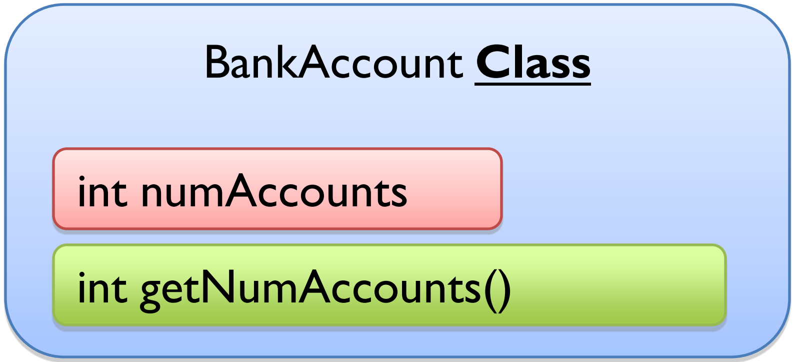
Instance Methods

- These are what you know so far…

- These define the operations you can perform on *objects* of a class.

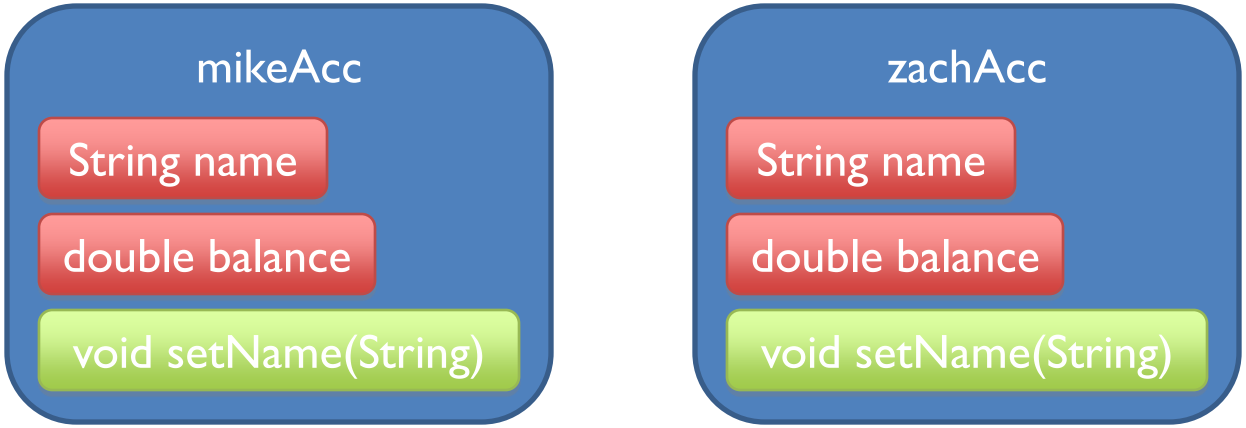
- Methods typically operate on the instance (non-static) fields of the class.
  - Each object has a “copy” of the method just as it has copies of the fields.

Static / Class Methods



- Static methods are shared by all objects of the class

- One copy for all objects



Static Methods

To define a class method, add the keyword **static** to its definition.

```
public class BankAccount {
    private static int numAccounts = 0;
    …

    public static int getNumAccounts() {
      return numAccounts;
    }
}
```

Calling Static Methods

```
public class BankAccount {
    private static int numAccounts = 0;
    …
    public static int getNumAccounts() {
     return numAccounts;
    }
}
```

```
BankAccount mikeAcc = new BankAccount(“Mike”, 100);
System.out.println(mikeAccount.getNumAccounts()); //1

BankAccount zachAcc = new BankAccount(“Zach”, 20);
System.out.println(mikeAccount.getNumAccounts()); //2
System.out.println(BankAccount.getNumAccounts()); //2
```

Static Methods

- Static methods do not operate on a specific instance of their class
- Have access only to static fields and methods of the class
  - Cannot access non-static ones

Static Methods Limitations

```
public class BankAccount {
    private static int nextAccountNum = 0;
    private int accountNum;
    …
    public static int getAccountNum() {
     return accountNum;
    }
}
```

Illegal, cannot access non-static field from static method

More Static Methods

- Static methods are also used when you need to define a method on 2 objects.

```
public static BankAccount greaterBalance
          (BankAccount ba1, BankAccount ba2)
{
    if (ba1.balance() >= ba2.balance())
     return ba1;
    else
     return ba2;
}
```

Static Method Examples

- For methods that use only the arguments and therefore do not operate on an object

  ```
  public static double pow(double b, double p)
  // Math class, takes b to the p power
  ```

- For methods that only need static data fields
- We **HAVE TO** use the static key word on the main method in the class that starts the program
  - No objects exist yet for the main method to operate on!

The `final` keyword

- Sometimes you will declare and initialize a variable with a value that will never change.

- To prevent any accidental changes, Java provides you with a way to fix the value of any variable by using the **`final`** keyword when you declare it.

The `final` keyword

- We declared `PI` as

  ```
  public static double PI = 3.14159;
  ```

  but this does not prevent changing its value:

  ```
  MyMath.PI = 999999999;
  ```

- We use keyword **`final`** to denote a constant:

  ```
  public static final double PI = 3.14159;
  ```

- Once we declare a variable to be **`final`**, it's value can no longer be changed!

Final References

- Consider this final reference to a `Point`:

```
public static final Point ORIGIN =
                            new Point(0,0);
```

- This prevents changing the reference `ORIGIN`:

```
      MyMath.ORIGIN = new Point(3, 4);
```

- BUT! You can still call methods on ORIGIN that change the state of ORIGIN.

```
          MyMath.ORIGIN.setX(4);
```