Creating a Mutable Class

Based on slides by Prof. Burton Ma

Mutable Classes

- A mutable class can change how its state appears to clients
  - Recall that immutable classes are generally easier to implement and use
  - So why would we want a mutable class?
    - Because you need a separate immutable object for every value you need to represent
      - Example is String concatenation

Reading a Text File into a String

```
BufferedReader in =
    new BufferedReader(new FileReader(file));
String contents = "";
while (in.ready()) {
    contents = contents + in.readLine();
}
```

creates a new String object
to perform the concatenation
each iteration of the loop

Reading a Text File into a StringBuilder

```
BufferedReader in =
    new BufferedReader(new FileReader(file));
StringBuilder contents = new StringBuilder();
while (in.ready()) {
    contents.append(in.readLine());
}
```

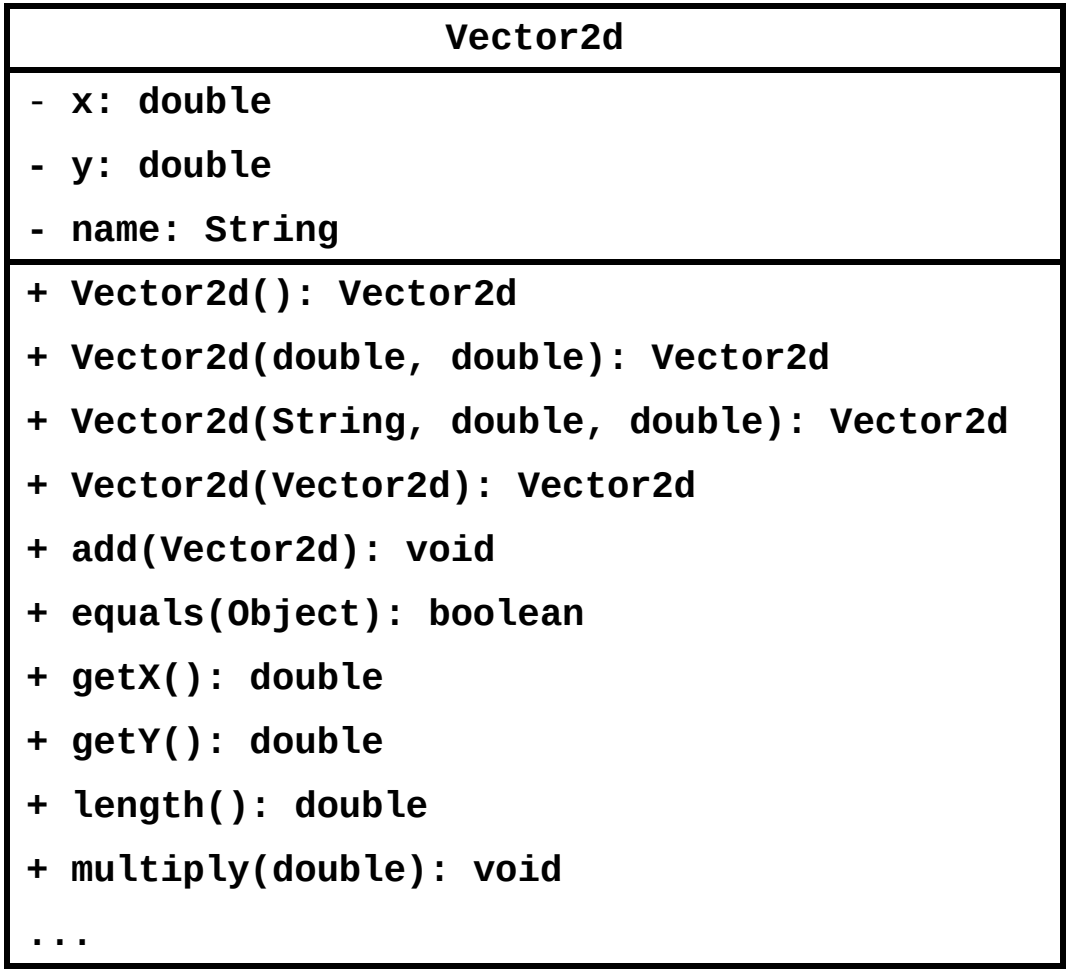
new String not created
for each iteration

Example Mutable class

- Create a class to represent 2-dimensional vectors

What Can Mathematical Vectors Do?

- add
- subtract
- multiply by scalar
- set coordinates
- get coordinates
- construct
- equals
- toString



Constructors

- Recall that the role of the constructor is to initialize the attributes of a new object
  - For **`Vector2D`** we need to initialize **`x`**, **`y`**, and **`name`**
- We have 4 overloaded constructors

`Vector2D()` Create the vector (0, 0) with no name.
`Vector2D(double x, double y)` Create the vector (x, y) with no name.
`Vector2D(String name, double x, double y)` Create the vector (x, y) with the given name.
`Vector2D(Vector2D other)` Create a new vector that is equal to the given vector.

Constructors

```
public Vector2D() {
  this.x = 0;
  this.y = 0;
  this.name = null;
}

public Vector2D(double x, double y) {
  this.x = x;
  this.y = y;
  this.name = null;
}
```

Constructors

```
public Vector2D(String name, double x, double y) {
  this.x = x;
  this.y = y;
  this.name = name;
}

public Vector2D(Vector2D other) {
  this.x = other.x;
  this.y = other.y;
  this.name = other.name;
}
```

Avoiding Code Duplication

- Notice that the constructor bodies are almost identical to each other
- Whenever you see duplicated code you should consider moving the duplicated code into a method
- In this case, one of the constructors already does everything we need to implement the other constructors…

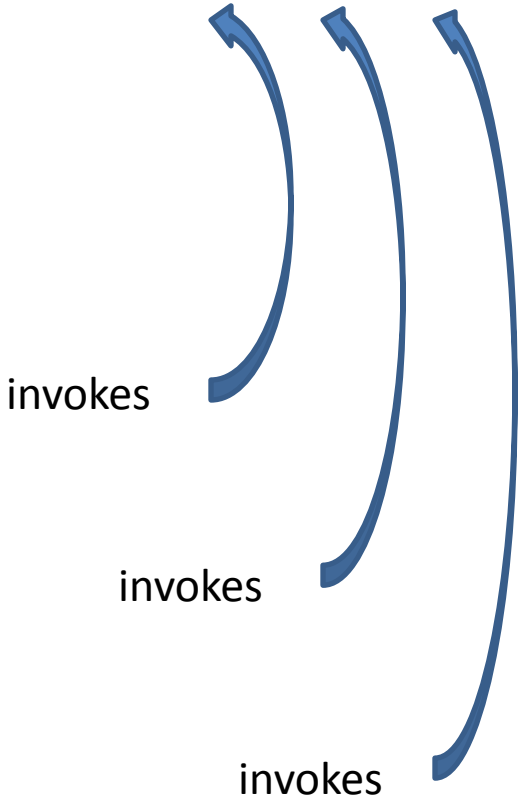
Constructors

```
public Vector2D(double x, double y, String name) {
  this.x = x;
  this.y = y;
  this.name = name;
}

public Vector2D() {
  this(0, 0, null);
}

public Vector2D(double x, double y) {
  this(x, y, null);
}

public Vector2D(Vector2D other) {
  this(other.x, other.y, other.name);
}
```



Constructor Chaining

- When a constructor invokes another constructor it is called *constructor chaining*
- To invoke a constructor in the same class you use the **`this`** keyword
  - If you do this then it must occur on the first line of the constructor body

Copy Constructor

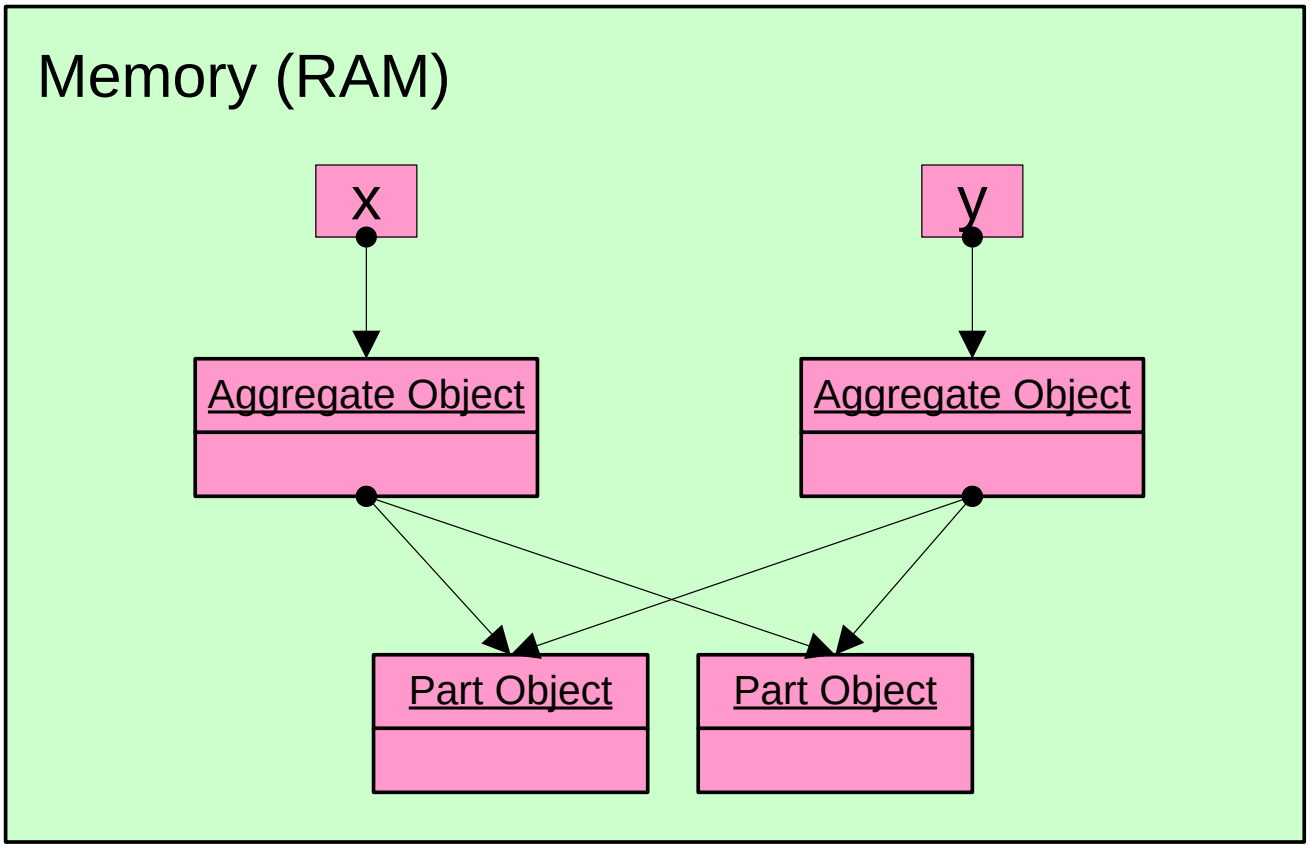
- The *copy constructor* is a notable overload
  - For a class x the copy constructor looks like

```
public X(X x)
```

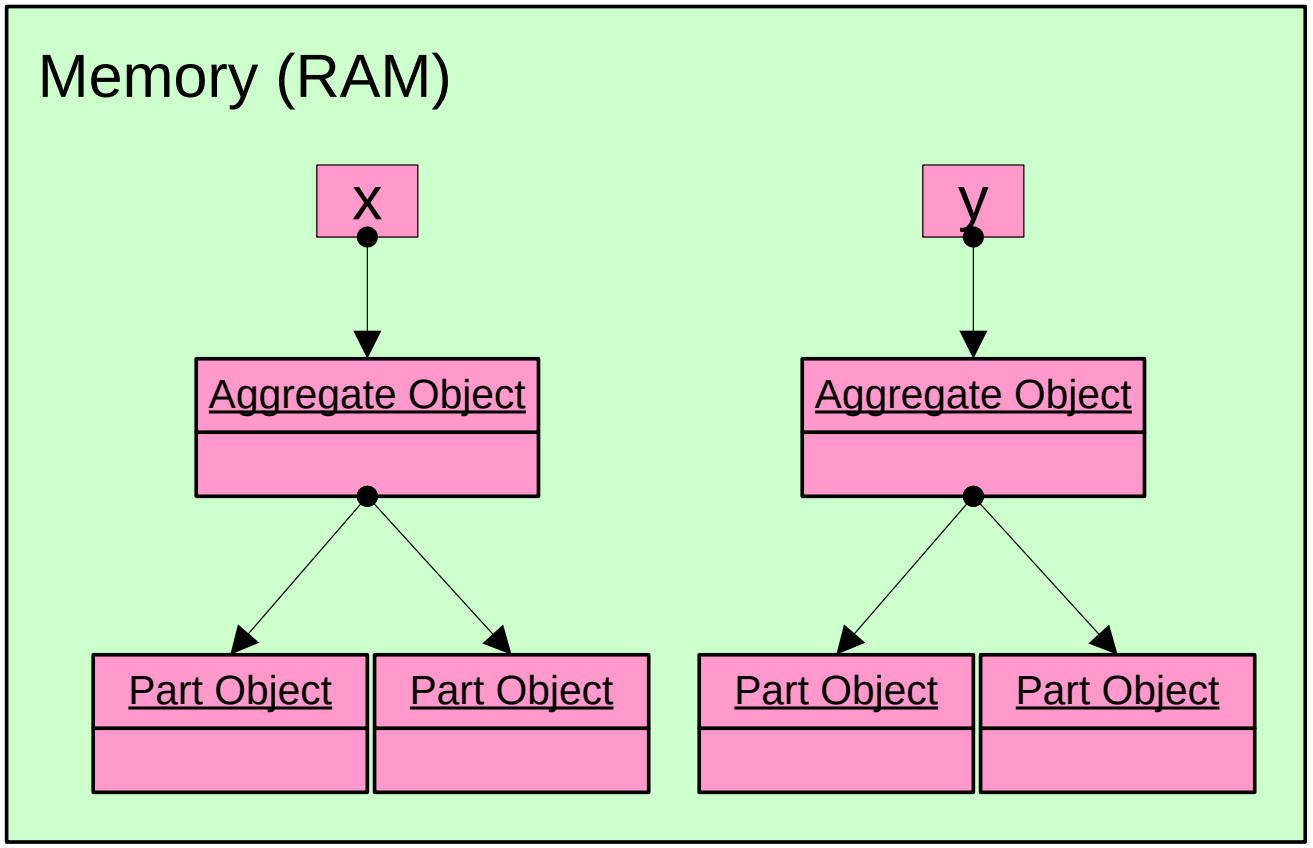
- A copy constructor creates an object by copying another object of the same type
  - You don't need (and should not declare) a copy constructor for immutable types

[AJ p 301-307]

Reminder: Shallow Copy



Reminder: Deep Copy



Accessor Methods

- Recall that accessor methods return information about the state of the object
  - For **`Vector2D`** we need to return information about **`x`**, **`y`**, and **`name`**
- We have 3 accessor methods

`double getX()` Get the x coordinate of the vector.
`double getY()` Get the y coordinate of the vector.
`String getName()` Get the name of the vector.

Accessor Methods

```
public double getX() {
  return this.x;
}
```

```
public double getY() {
  return this.y;
}
```

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

Mutator Methods

- Recall that mutator methods allow a client to manipulate the state of the object
  - For **`Vector2D`** we need to allow the client to manipulate **`x`**, **`y`**, and **`name`**

Mutator Methods

- We have 5 mutator methods

`void setX(double x)` Set the x coordinate of the vector.
`void setY(double y)` Set the y coordinate of the vector.
`void setName(String name)` Set the name of the vector.
`void set(double x, double y)` Set the x and y coordinate of the vector
`void set(String name, double x, double y)` Set the name, x, and y coordinate of the vector

`setX()`, `setY()`, and `set()`

```
public void setX(double x) {
  this.x = x;
}

public void setY(double y) {
  this.y = y;
}

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

public void set(double x, double y) {
  this.setX(x);
  this.setY(y);
}

public void set(String name, double x, double y) {
  this.setName(name);
  this.set(x, y);
}
```

Equals

- Recall that most value type classes will want their own version of **`equals`**
  - We shall say that two vectors are equal if their **`x`**, and **`y`** coordinates are equal
    - i.e., two vectors might be equal even if their names are different

`boolean equals(Object obj)`
Compares two vectors for equality.

equals()

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }




  return eq;
}
```

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {



  }
  return eq;
}
```

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;



  }
  return eq;
}
```

This version works most of the time (except when it doesn't!)

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;
    eq = this.getX() == other.getX() &&
         this.getY() == other.getY();
  }
  return eq;
}
```

This version always works.

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {
    Vector2d other = (Vector2d) obj;
    eq = Double.compare(this.getX(), other.getX()) == 0 &&
         Double.compare(this.getY(), other.getY()) == 0;
  }
  return eq;
}
```

== vs Double.compare

- The issue here is quite subtle
- If you use == to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`NaN == NaN`** is always **`false`**

  - NaN means “not a number” and is used to represent a mathematically undefined number
    - Such as occurs when you divide zero by zero
    - The behavior of NaN is defined in the IEEE 754 standard for floating point arithmetic (i.e., this is not just a Java issue)

== vs Double.compare

- If you use == to compare the coordinates then all hash based collections and all sets will behave strangely with vectors having NaN as a component

```
Set<Vector2D> set = new HashSet<Vector2D>();
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
set.add(u);
set.add(v);
System.out.println(set.size());             // prints 2
```

- Sets are supposed to reject duplicate elements but there are 2 identical vectors in **`set`**
  - Occurs because **`Set`** uses **`equals`** to check for duplicates

== vs Double.compare

- If you use **`Double.compare`** to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`true`** because **`Double.compare`** is implemented to allow for equality of **`NaN`**

- Checking for equality of **`NaN`** can be useful when trying to track down errors in computations
- Also the hash based collections and sets will work as expected

== vs Double.compare

- There is a side effect of using **`Double.compare`** to compare the coordinates

```
Vector2D u = new Vector2D(0.0, 1.0);    // (0.0, 1.0)
Vector2D v = new Vector2D(-0.0, 1.0);   // (-0.0, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`Double.compare`** considers 0.0 and -0.0 to be unequal

  - Can you see how to implement equals to allow for equality of **`NaN`** and equality of 0.0 and -0.0?

== vs Double.compare

- The real issue here is that floating point arithmetic is tricky and affects every programming language
- A good starting point for learning more about some of the issues involved
  - http://floating-point-gui.de/

Observe That...

- Instead of directly using the attributes, we use accessor methods where possible
  - This reduces code duplication, especially if accessing an attribute requires a lot of code
  - This gives us the possibility to change the representation of the attributes in the future
    - As long as we update the accessor methods (but we would have to do that anyway to preserve the API)
  - For example, instead of two attributes `x` and `y`, we might want to use an array or some sort of `Collection`
- The notes [notes 2.3.1] call this *delegating to accessors*

Observe That...

- Instead of directly modifying the attributes, we use mutator methods where possible
  - This reduces code duplication, especially if modifying an attribute requires a lot of code
  - This gives us the possibility to change the representation of the attributes in the future
    - As long as we update the mutator methods (but we would have to do that anyway to preserve the API)
  - For example, instead of two attributes `x` and `y`, we might want to use an array or some sort of `Collection`
- The notes [notes 2.3.1] call this *delegating to mutators*