CLASSES

Edwin Chan

CPSC 233 W16 T01/T05

Main() + Class

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Complex a, b;

		// Initializes real and imag to 0.0
		a = new Complex();

		// Initializes real to -3.0 and imag to 4.3
		b = new Complex(-3.0, 4.3);
	}
}
```

This is a ***main() function***. Main() functions are entry points into your program, they tell the interpreter where to start when it runs your program. In our examples and most of your assignments, your main program code will go here. The following slides will show you how to make use of other Classes inside your code here.

Complex.java

```
// complex numbers
public class Complex {
	private double real, imaginary;

	public Complex(double rVal, double iVal){
		real = rVal;
		imag = iVal;
	}

	public Complex(){
		real = 0.0;
		imag = 0.0;
}
```

On this side, there will always be a Class which the main() function interacts with.

Constructors

MainProgram.java

```
public class MainProgram{
      public static void main(String[] args){
            Complex a, b;

            // Initializes real and imag to 0.0
            a = new Complex();

            // Initializes real to -3.0 and imag to 4.3
            b = new Complex(-3.0, 4.3);
      }
}
```

Complex.java

```
// complex numbers
public class Complex {
      private double real, imaginary;

      public Complex(double rVal, double iVal){
            real = rVal;
            imag = iVal;
      }

      public Complex(){
            real = 0.0;
            imag = 0.0;
      }
}
```

signature matching

Same *constructor* name, so they are *overloaded*.

signatures

A ***signature*** is the *method name* plus the *parameter list*. It DOES NOT include the return type.

```
public int exampleMethod(String someParam, int otherParam){}
//The signature here is exampleMethod(String, int) ONLY.
```

Overloading is when you have two (or more) constructors or methods, and they have the *same name but different parameter lists*.

Signature matching is when you call a constructor or method, and it looks for the matching one with the same signature.

Class vs Instance

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Complex a, b;

		// Initializes real and imag to 0.0
		a = new Complex();

		// Initializes real to -3.0 and imag to 4.3
		b = new Complex(-3.0, 4.3);
	}
}
```

Complex.java

```
// complex numbers
public class Complex {
	private double real, imaginary;

	public Complex(double rVal, double iVal){
		real = rVal;
		imag = iVal;
	}

	public Complex(){
		real = 0.0;
		imag = 0.0;
}
```

a and *b* are *instances* of the *Complex class*

a and *b* are *objects* of *type Complex*

Class vs Instance

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Point a, b;
		// Initializes points a(1,4) and b(7,10)
		a = new Point(1,4);
		b = new Point(7,10);

		// Let's call some instance methods
		a.setX(5); // a(5,4)   b(7,10)
		b.setX(2); // a(5,4)   b(2,10)
		b.setY(3); // a(5,4)   b(2,3)
	}
}
```

Notice how Point instances *a* and *b* are independent of each other. They are separate objects, each with their own x and y values (instance variables).

Point.java

```
// points
public class Point {
	private double x, y;
	private static int instances = 0;

	public void setX(double value) {
		x = value;
	}

	public void setY(double value) {
		y = value;
	}

	public static int numberInstances(){
		return instances;
	}
}
```

When we want to make changes to instance variables, we call instance methods.

Class vs Instance

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Point a, b, c;
		// Let's make a few Point objects
		a = new Point(1,4);
		b = new Point(7,10);
		c = new Point(5,-2);

		// How many Points do we have now?
		int numberOfPoints = Point.numberInstances();
	}
}
```

See how we call the class method using the class name (Point) and not one of the instances (a, b, c)?

Point.java

```
// points
public class Point {
	private double x, y;
	private static int instances = 0;

	public Point(double xVal, double yVal){
		this.x = xVal;
		this.y = yVal;

		// Add one to the number of Points
		instances++;
	}

	public static int numberInstances(){
		return instances;
	}
}
```

When we want to access class variables, we call class methods. Notice the **static** keyword in both.

Class vs Instance

MainProgram.java

```
public class MainProgram{
      public static void main(String[] args){
            // here we set aside space for three objects
            // of type Point, but we haven't actually made
            // them yet, that's why we still have 0 Points
            Point a, b, c;
            int numberOfPoints;

            numberOfPoints = Point.numberInstances(); // 0
            // Let's make a few Point objects
            a = new Point(1,4);
            b = new Point(7,10);
            numberOfPoints = Point.numberInstances(); // 2
            c = new Point(5,-2);
            numberOfPoints = Point.numberInstances(); // 3

            a = b = c = null;
            // we delete all the points
            // assume garbage collection is done here
            // we will cover that soon
            numberOfPoints = Point.numberInstances(); // 0
      }
}
```

Even if you delete every instance of Point, you can still access the class methods and class variables.

Point.java

```
// points
public class Point {
      private double x, y;
      private static int instances = 0;

      public Point(double xVal, double yVal){
            this.x = xVal;
            this.y = yVal;

            // Add one to the number of Points
            instances++;
      }

      public static int numberInstances(){
            return instances;
      }
}
```

Static means they are permanent, always there throughout your program's execution.

Access Control (public/private, get/set)

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Student zelda = new Student("Princess", "Zelda");

		// WRONG: directly accessing a private variable
		// This will not work, you will not even be able
		// to see the variable, much less change it.
		zelda.grade = 100;

		// CORRECT: use public get/set methods
		// get/set methods AKA accessor/mutator methods
		zelda.setGrade(100);
		System.out.println(zelda.getGrade()); // 100
	}
}
```

ERROR! Variable is private.

OK! setGrade(int) and getGrade() are public instance methods.

Student.java

```
// students
public class Student{
	private String firstName;
	private String lastName;
	private int grade;

	public Student(String first, String last)
	{
		this.firstName = first;
		this.lastName = last;
		this.grade = 80;
	}

	public void setGrade(int newGrade){
		this.grade = newGrade;
	}

	public int getGrade(){
		return this.grade;
	}
}
```

OK! The variable is accessed directly, within the class itself.

Private: this cannot be accessed directly except by the class's own methods and constructors.

The *this* Keyword (self reference)

MainProgram.java

```
public class MainProgram{
      public static void main(String[] args){
            Point origin = new Point(0, 0);
      }
}
```

Imagine going "inside" the code which makes up the origin object

If you imagine "going inside the *origin* object" or "looking at the code inside the *origin* object", then *this* inside *origin*'s code refers to *origin*. *this.x* here is really referring to the x field/variable in the Point instance called *origin,* except we use *this* because we don't we don't have any other way to reference the object *(origin* is not defined in Point.java".

Point.java

```
// points
public class Point{
      private double x;
      private double y;

      public Point(double xVal, double yVal)
      {
            this.x = xVal; // best way
            this.y = yVal; // best way
      }
}
```

The *this* keyword is a ***reference*** **to a particular instance** of a class, by the instance itself. For example, we make a Point *instance* called *origin*. Whenever we invoke a constructor or instance method for *origin*, the *this* refers to the instance *origin*.

Example on the next page >>>

Point origin = new Point(0, 0);

constructor for *origin*

this = origin

Point *origin*

```
public class Point{
	private double x;
	private double y;

	public Point(double xVal, double yVal)
	{
		this.x = xVal; // best way
		this.y = yVal; // best way
	}
}
```

Point pointTwo = new Point(5, 5);

constructor for *pointTwo*

this = pointTwo

Point *pointTwo*

```
public class Point{
	private double x;
	private double y;

	public Point(double xVal, double yVal)
	{
		this.x = xVal; // best way
		this.y = yVal; // best way
	}
}
```

Point pointThree = new Point(1000, 1000);

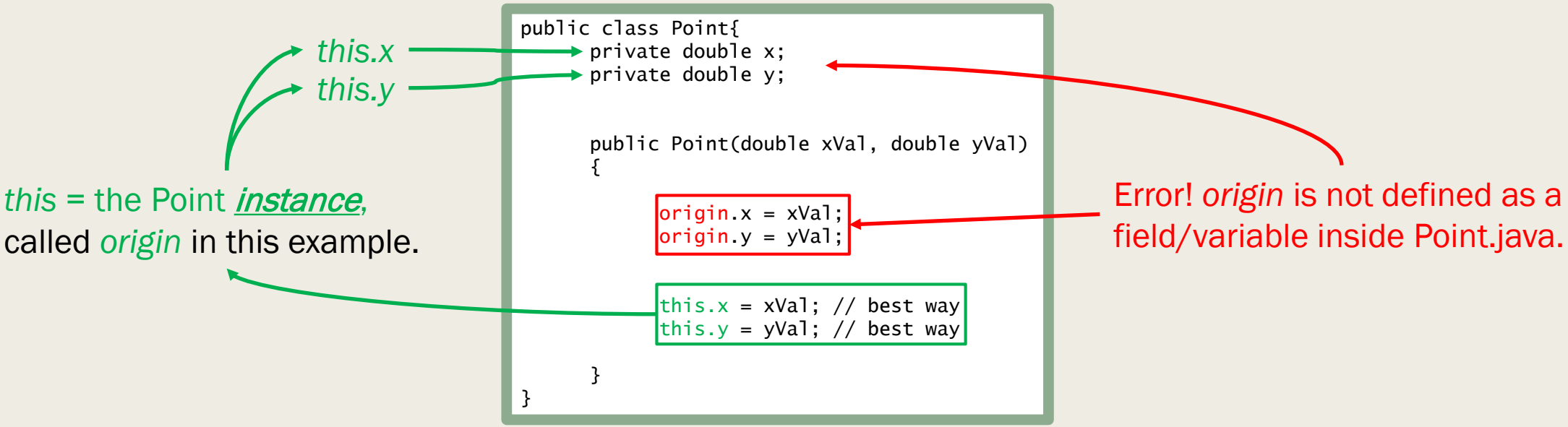
constructor for *pointThree*

this = pointThree

Point *pointThree*

```
public class Point{
	private double x;
	private double y;

	public Point(double xVal, double yVal)
	{
		this.x = xVal; // best way
		this.y = yVal; // best way
	}
}
```


this.x
this.y
this = the Point instance,
called origin in this example.
public class Point{
private double x;
private double y;
public Point(double xVal, double yVal)
{
origin.x = xVal;
origin.y = yVal;
this.x = xVal; // best way
this.y = yVal; // best way
}
}
Error! origin is not defined as a field/variable inside Point.java.

The *this* Keyword (self reference)

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){
		Point origin = new Point(0, 0);
	}
}
```

Point.java

```
// points
public class Point{
	private double x;
	private double y;

	public Point(double xVal, double yVal)
	{
		this.x = xVal; // best way
		this.y = yVal; // best way
	}

	public Point(double xVal, double yVal)
	{
		x = xVal; // okay but not as clear
		y = yVal; // okay but not as clear
	}

	public Point(double x, double y)
	{
		x = x; //wrong
		y = y; //wrong
	}
}
```

The variables are clearly differentiated, with both the *this* keyword and the different variable names.

Not using the *this* keyword, but the variable names are different, so it still works.

ERROR! You are assigning the variables from the parameters to themselves. If you always use the *this* keyword, you can prevent this from ever happening. (good coding practice)

The *this* Keyword (calling constructors)

MainProgram.java

```
public class MainProgram{
      public static void main(String[] args){

            Point origin = new Point();

            Point pointFarAway = new Point(1000, 1000);
      }
}
```

Point.java

```
// points
public class Point{
      private double x;
      private double y;
      private int pointID;
      private boolean visible;

      public Point(double xVal, double yVal)
      {
            this.x = xVal;
            this.y = yVal;

            this.pointID = 0;
            this.visible = true;
      }

      public Point()
      {
            this.x = 0; //
            this.y = 0; //

            this.pointID = 0;
            this.visible = true;
      }
}
```

Problem: We are duplicating code for each constructor.

The *this* Keyword (calling constructors)

MainProgram.java

```
public class MainProgram{
	public static void main(String[] args){

		Point origin = new Point();

		Point pointFarAway = new Point(1000, 1000);
	}
}
```

Point.java

```
// points
public class Point{
	private double x;
	private double y;
	private int pointID;
	private boolean visible;

	public Point(double xVal, double yVal)
	{
		this.x = xVal;
		this.y = yVal;

		this.pointID = 0;
		this.visible = true;
	}

	public Point()
	{
		this(0, 0);
	}
}
```

call another constructor in the Point class with the signature *Point(double, double)*

The *this* keyword can also be used within one constructor to **invoke another constructor**. In this example, we make two points using two constructors. Notice the different signatures:

Point()

Point(double, double)