I’m Gary, and I’ll be volunteering

I’ve loved computers since I was very little. My first computer was an Apple][(1977).

The Apple got me interested in programming. It was satisfying to get the computer to do something. I learned BASIC, Pascal, and 6502 assembly language.

I have been in the industry for 26 years. I used to work on Adobe Flash, which was used for many early Internet games and cartoons.

On the job, it was satisfying to know lots of people used software I worked on, and even made their own creative projects using our software.

zendesk

Today, I’m a Software Architect at Zendesk, a maker of customer support software. Our main office is in San Francisco by Civic Center BART.

Calling Methods without Parameters

What is a method?

A method is an **action** defined for a class that all instances of that class (objects) will support.

Methods can:

1. Provide access to an attribute of an instance
2. Update an attribute of an instance
3. Do something new and interesting with the information stored in an instance

Methods are called using the "." operator, which allows access to the public methods of a class.

An example… what number is printed?

```
// Dog.java

public class Dog {
	private int age; // an attribute

	public Dog(int dogAge) { // constructor
		age = dogAge;
	}

	// a method that updates an attribute.
	// returns nothing
	public void makeOlder(int years) {
		age += years;
	}

	private int dogYears() { // an internal method
		return 7*age;
	}

	// a method that retrieves an attribute
	public int getAge() {
		return dogYears();
	}
}
```

```
// TestDog.java

public class TestDog {

	public static void main(String[] args) {

		Dog goodBoy = new Dog(5);
		goodBoy.makeOlder(2);
		int age = goodBoy.getAge();
		System.out.println(age);
	}
}
```

Method declarations

Method declarations, such as `public void makeOlder(int years) { … }`

1. **Define whether the method is accessible to the outside world** (public / private)
   a. Public methods are available externally (e.g. `goodBoy.getAge()`) while private methods are not (calling `goodBoy.dogYears()` in `main` will cause an error)
2. **Determine what the method returns**
   a. Void methods return nothing
   b. String methods promise to return Strings, int methods to return ints
3. **Defines the variables (parameters) passed to the method**
   a. To be described in the next section
4. **Define the body of the method**
   a. The body is the statements of code that will execute when the method is called.

Abstraction – keeping things simple

One of the core concepts in computer science is **abstraction.** Abstraction means that you only need to understand how to interact with an object–you **don't need to understand how the code is actually implemented behinds the scenes.**

E.g. as a user, I should be indifferent between the following implementations:

Option 1

```
private int dogYears() {
    return 7*age;
}

public int getAge() {
    return dogYears();
}
```

Option 2

```
public int getAge() {
    return (age + age +
            age + age +
            age + age +
            age);
}
```

Option 3

```
public int getAge() {
    return 7*age;
}
```

The power of abstraction

Abstraction accomplishes two things:

1. It keeps things simple, minimizing what you need to know to write a program
2. It makes it possible for the class owner to change the technical implementation of the method without impacting its use
    a. e.g., option 3 may be faster for a computer to calculate than option 2… the programmer may want to switch their implementation from 2 to 3. Abstraction means that the user won't notice a difference (besides faster code)

NullPointerException

A variable `Dog dog;` **points to** an instance of `class Dog`.

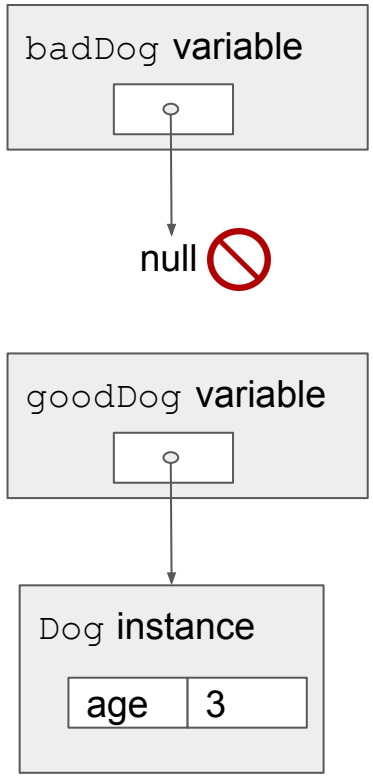
It starts out not pointing to any `Dog`, with the special value `null`.

You have to use `new Dog` to construct a Dog instance that the variable can point to.

If you don't **initialize** a variable to point to a `Dog` instance, and you try to call a method, `NullPointerException` will be thrown.

```
Dog badDog; // badDog == null
badDog.getAge(); // throws NullPointerException

Dog goodDog = new Dog(3); // goodDog points to instance
goodDog.getAge(); // no problem
```



Instance and static methods

Instance methods act upon instances of a class. We first create an instance and then call on one of its instance methods.

```
String t = "blue”;

t.substring(0,2); // "bl”
```

Static methods aren’t bound to a particular instance of a class. They are called by naming a class following by the dot operator:

```
String.valueOf(1234); // "1234”
```

So what does `public static void` mean?

Methods with Parameters

Write a function that prints a greeting

```
public class Person {
    String name;
    public Person(String personName) {
        name = personName;
    }

    // Greeting
    public void greet() {
        System.out.println(name + " says: Hello, world!”);
    }
}
```

Use parameters!


```
// Person.java
public class Person {
     private String name;

     public Person(String personName) {
          name = personName;
     }

     // Greeting
     public void greet() {
          System.out.println(name + " says: Hello, world!");
    }

     // Greet a particular person
     public void greet(String otherName) {
          System.out.println(name + " says: Hello, " + otherName + "!”);
    }
}
```

Calling methods with parameters

```
// TestPerson.java
public class TestPerson {
      public static void main(String[] args) {
            Person amy = new Person("Amy");
            amy.greet("Ted"); // Prints "Amy says: Hello, Ted!"
            amy.greet("Thursday"); // Prints "Amy says: Hello, Thursday!"

            Person bob = new Person("Bob");
            bob.greet("Amy"); // Prints "Bob says: Hello, Amy!"
      }
}
```

Definitions

Formal Parameter (parameter) -The variable declared in the method header

```
public void greet(String name)
```

Actual Parameter (argument) - The value passed in a method call

```
amy.greet("Ted");
```

Method Overloading

Overloaded methods are two or more methods in the same class that have the same name but different parameters.

```
// Person.java
public class Person {
    private String name;

    public Person(String personName) {
        name = personName;
    }

    // Greeting
    public void greet() {
        System.out.println(name + " says: Hello, world!");
    }

    // Greet a particular person
    public void greet(String otherName) {
        System.out.println(name + " says: Hello, " + otherName + "!”);
    }

    // Greet a number
    public void greet(int aNumber) {
        System.out.println(name + " says: How are you, " + aNumber + "?");
    }
}
```

Calling Overloaded Methods

```
// TestPerson.java
public class TestPerson {
    public static void main(String[] args) {
        Person amy = new Person("Amy”);
        amy.greet(); // Prints "Amy says: Hello, world!”
        amy.greet("Ted”); // Prints "Amy says: Hello, Ted!”
        amy.greet(12); // Prints "Amy says: How are you, 12?”
    }
}
```

Method signatures

In Java, the method signature is the method name and just the types of the parameters. It doesn't include return type, parameter names, or public/private. For the method declaration

```
public void addCustomer(String name, String address, int age) {...}
```

The method signature is:

```
greet(String, String, int)
```

A Java class cannot have two methods with the same signature. The Java compiler needs method signatures to figure out which overloaded method to call.

OK, signatures different:

```
int square(int x) { return x*x; }
double square(double x) { return x*x; }
```

Error, signatures are the same:

```
int square(double x) { return x*x; }
double square(double y) { return y*y; }
```

Practice!

More Exercises

1. Write three different implementations for a single static method **triple**. The method should take in a number N and return triple its value. Should the method signatures differ across your implementations? Why or why not?
2. If one of these methods was implemented as part of the class Number, how would it be called?
3. Now create a Number class with a constructor that accepts a number and stores it in a private variable. Create a **triple** instance method that returns three times that value. How does this instance method differ from the static method you defined earlier?

Turtle Class

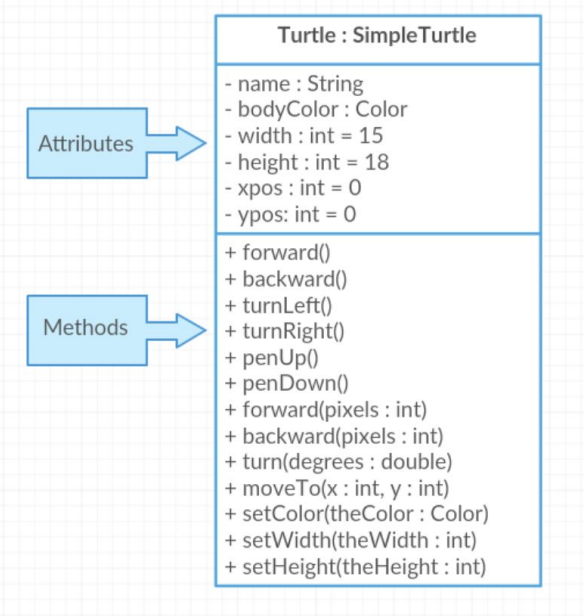


Figure 1: Turtle Class Diagram

Color Class

- This is a standard Java class
- This is an example of abstraction in action!
- Color has overloaded constructors:
  `Color(float r, float g, float b)`
  `Color(int r, int g, int b)`
- Color has instance methods:
  `int getRGB()`
- Color has static methods too:
  `static int HSBtoRGB(float hue, float saturation, float brightness)`
- Color has static member variables for common colors, e.g., `Color.red`, `Color.gray`, `Color.white`

Extra Challenge: Make a cool design!