Object Oriented Programming with Java

Classes & Objects

Classes

- Just as we have many number values that are elements of the int type, we can create many objects that are elements of some specific class type.
- A class is a ***blueprint*** for making multiple values (objects) that have the same structure/methods.
- We call a class type a **reference type**, because we deal with references pointing to the objects stored in memory, versus dealing with the actual values for primitive types.

An object is a *value*

An **object** is an ***instance*** of a class (a specific value of that reference type).

From one class definition we can make many unique objects, each with their own state (values).

Objects are just like other values: we can create variables to hold objects, create arrays to hold many same-type object values, and even create objects just to use within an expression.

A class is a *type*

A **class** defines a new variable **type**, from which we can create **objects** (values)

Java provides many built-in classes/types like `Scanner`, `File`, `PrintWriter`, etc.

But we can create our own classes/types and give them any name we like. For example, `Student`, `Course`, `Point`, etc.

A class definition specifies:

- what **attributes** (data/variables) the class has
- what **behaviors** (functions/methods) the class can exhibit

Class Components

The class's name is the newly defined type's identifier.

Fields: declaring a variable directly in a class represents an **instance variable**: each object will have its own value for each instance variable.

Behaviors: a **method** defined directly in a class can be called on any object of that class, using that object's instance variables.

Class Example – Knapsack class

Fields: `size`, `items`, `isFull` are *instance variables*: each Knapsack object will have its own values for each.

Behaviors: `getItem` is a *method* that can be called on a `Knapsack` object, using that object's instance variables.

```
class Knapsack {
    int size;
    String[] items;
    boolean isFull;

    String getItem(int i) {
        return items[i];
    }
}
```

Class syntax

- A class definition is: any modifiers, the class keyword, followed by the class's identifier, followed by curly braces. In the curly braces two different sorts of definitions are allowed:

```
modifiers class identifier {
    field definitions
    method definitions
}
```

- ***field definitions:*** modifiers, type, identifier. Declares fields (variables) associated only with this class.
- ***method definitions:*** modifiers, return type, identifier, parameter list, body. Declares methods associated only with this class.
- **simplifications**: we are ignoring syntax for inheritance and interface implementation for now – there is more to a class definitions syntax than shown above.

Class Syntax Notes

Modifiers: modifiers (like public, private, protected, final) can make substantial changes to the meaning of the variables and methods of a class. We will study these in more details throughout the semester.

Quick Tour:

- → public, private, protected: controls who can access it
- → static: controls whether it's a single shared thing or tied to specific object
- → final: disallows further changes
- → abstract: thing can be extended, but not directly used
- → synchronized, volatile: used in threading

Ordering: Although variables and methods can be listed in any mixed order, it is quite common to put all variables first, followed by all methods (and constructor methods first among methods).

Object Creation (instantiation)

To create (instantiate) an object, we call the class's constructor method.

Syntax: the `new` keyword, the class type (the class's name), and an argument list in parentheses for the constructor method.

- Example: `new Knapsack()`
- The default constructor has no parameters, but we can create our own constructors that do much more (discussed later).

Semantics: Java uses the constructor method and makes space in memory for another object (space for all instance variables).

- Initial values are set up according to the constructor method's code. A ***reference*** to this spot in memory is the result of evaluating this constructor call (hence the name "reference type").

Object Example

- knap is a reference to an object having the type Knapsack (it is a Knapsack value)
- we access/update an instance variable by:
  `objExpr.instVarName`
  ex: `knap.size`
- we ask an object to call its method on itself by:
  `objExpr.methodName(args)`
  ex: `knap.getItem(1)`

```
// create an object (call constructor)
Knapsack knap = new Knapsack();

// manipulate it
knap.size = 2;
knap.items = new String[ knap.size ];
knap.items[0] = "map";
knap.items[1] = "nutella";
knap.isFull = true;

// interact with it: call methods
String item = knap.getItem(1);
System.out.println(knap.size);
System.out.println(item);
```

Object Uniqueness

- Multiple objects of one type can be created that are distinct. Each will occupy a separate spot in memory. For example, **knap1** and **knap2** are references to two different objects
- But aliasing does not create a new object, just a reference (e.g. **knap3**)

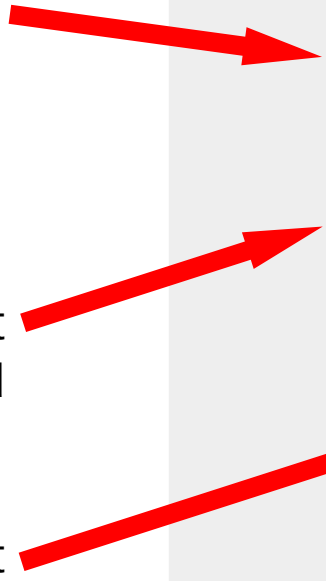
```
Knapsack knap1 = new Knapsack();
Knapsack knap2 = new Knapsack();
Knapsack knap3 = knap1;
knap1.size = 5;
knap2.size = 8;
knap3.size = 0;
System.out.println(knap1.size);
System.out.println(knap2.size);
```

What is printed?

Scope of Variables

There are three kinds of variables in Java:

- **Instance variables**: variables that hold data for an instance of a class. Each object has its own copy of all the instance variables of the class.
- **Class variables**: variables that hold data that will be shared among all instances of a class.
- **Local variables**: variables that pertain only to a block of code, i.e. inside a method or further nested like in a for-loop.



```
class Car {
    int price;

    static float gasPrice;

    void print() {
        boolean a;
        System.out.print(price);
    }
}
```

Objects are *values*

Objects are just values of a particular type (the given class type).

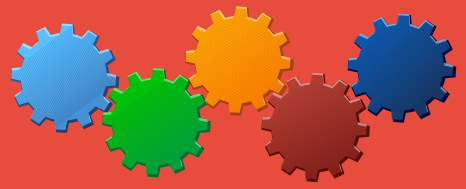
"Every expression has a type": class-definitions are types too!

Object-yielding expressions (yield a value of a reference type):

- constructor calls
- methods with a return type that is a classname

Nothing Special… Arrays of a reference type work just like arrays of primitive types. Expressions involving reference types are still just expressions. Variables can store reference types.

Practice Problems



Create a class for a Coordinate (an x and y value representing two dimensions).

- What instance variables should this class have?

Create a class for a Square.

- What instance variables should this class have?

Quick Note on Using Classes

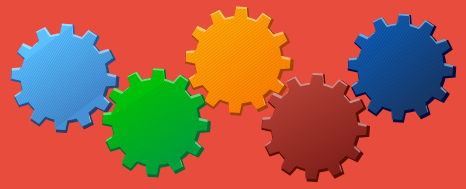
- We put one class definition in each file, and give the file the class's name: ClassName.java.
- For files in the same directory, we can just use another class by name:

TestKnap.java (in same directory as Knapsack.java)

```
class TestKnap {
    public static void main (String[]args){
        //We can just use the Knapsack class directly
        Knapsack knap1 = new Knapsack(3,new String[]);
        knap1.items[0] = "map";
    }
}
```

We will look at packages later, as a way to organize all these class files

Practice Problems



- Create a separate class, named `TestOtherClasses`. Assume it resides in the same directory as `Coordinate` and `Square`.
- In the main method:
  - Create two Coordinate objects, and give them distinct starting values.
    - Print out whether the first or second Coordinate object is further away from the origin (or they are the same). Use the method for these calculations.
  - Create a Square object. Print out its side length, area, and perimeter. Change its size, and then again print out its side length, area, and perimeter.
    - Is there any way we could keep ourselves from rewriting that printing code each time? What key concepts are involved?

Methods

Methods (brief review)

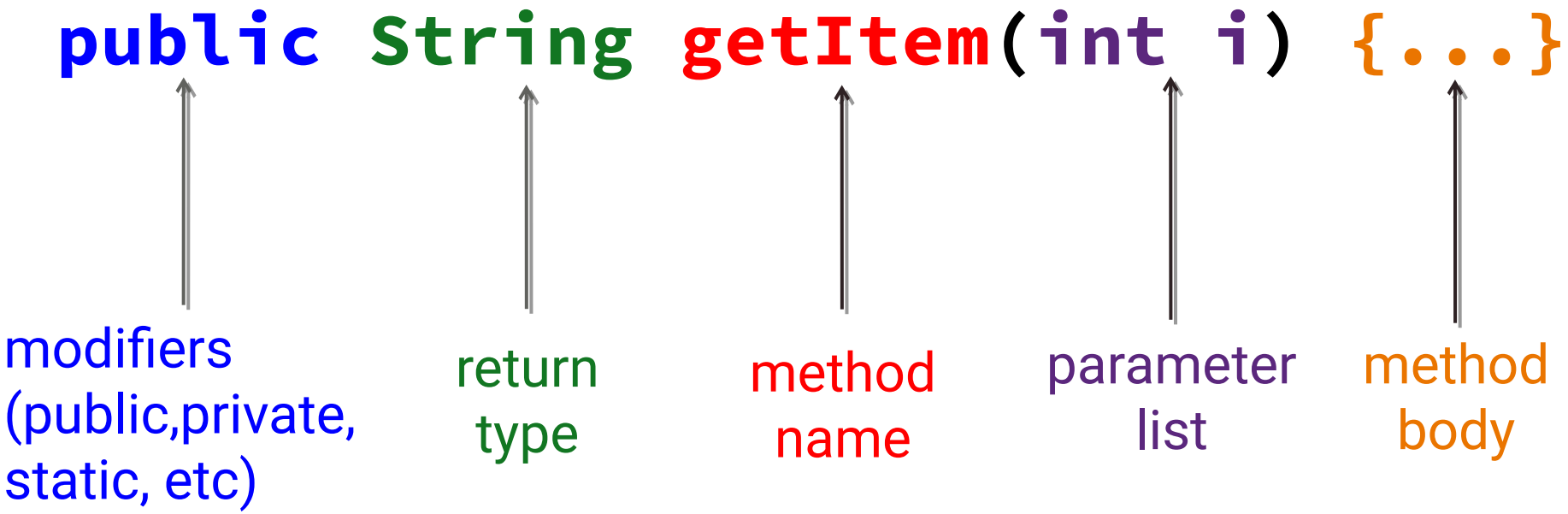
A method is a named block of code that can be called. It is like a function that is associated with a specific object or class.

Our first view of methods: defined in a class. Requires an object of that class type, and the object performs the call using its own instance variables.

→ *this is almost the whole story, but not quite!*

Java Methods in a nutshell

Method Signature: all the modifiers, type information (return type and parameters), and the name. Provides all details needed for interacting with/identifying the method.



Methods

When we define a method we must:

- ***state the return type***: what type of value will be returned? If no value should be returned, return type is listed as `void`.
- ***define the listing of parameters:*** types and names for values that are supplied (as an 'argument list') at each method call.
- ***write the method body*** (block of code) using parameters to perform some task and return a value of the indicated return type.

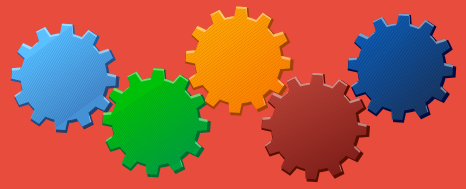
Method Example - `getItem`

- ***Modifier*: `public`.**
  (visible outside an object)
- ***Return type*: `String`.** the method must return a value, and it must be a String.
- ***Name*: `getItem`.**
- ***Parameter list*: `(int i)`.**
  variable declarations (without instantiations) that are in scope only in this method call
- **Method Body:** uses the object's `items` instance variable to find one String value and return it.

```
class Knapsack {
    public int size;
    public String[] items;
    public boolean isFull;

    public String getItem(int i) {
        return items[i];
    }
}
```

Practice Problems



Coordinate Class:

- Add a method that calculates the distance from the origin. (Use Pythagoras' Theorem).
  - use Math.sqrt and Math.pow. (java.lang.Math is always available)

Square Class:

- Add methods to calculate the area and the perimeter.

Knapsack Class:

- Add a method to check if the knapsack is full, and remove the isFull instance variable. (Why might this be a good idea?)

Constructor Method

A constructor method is a special method that is used to create a new object value of the class. It implicitly returns a reference to this new object.

Return Type: *not specified*, because the constructor always returns a reference to an object of the class type.

Method Name: always identical to the class name.

Parameters: Just like regular methods (can have zero or more). Often, parameters are (nearly) one-to-one matches of instance variables.

Body: chance to set initial values for instance variables, perform consistency checks, do any other involved setup work (including calling other methods)

Knapsack Example: Constructor

- **Parameter list:** two params, for size and starting items.
- **body:** initialized all instance variables (based on params).
- **Point of No Return!** A constructor doesn't need an explicit return statement to return the object, because a reference to the newly created object is implicitly returned.

```
class Knapsack {
    …
    public Knapsack (int s, String[] is) {
        size= s;
        items = is;
        isFull = (s==items.length);
    }
}
```

Default Constructors

Default Constructor: Java provides a default constructor definition when none is present in a class: there are no parameters, and all instance variables get default values: primitive types get `0`, `0.0`, `false`; reference types (class types) get the `null` value.

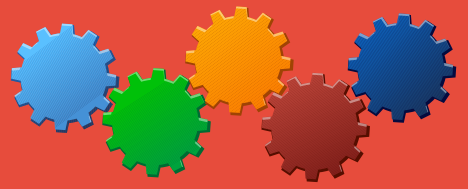
`null` is a special value for reference types only: A variable that has a value of `null` is pointing to no object in memory. Attempting to **dereference** that variable will result in a run-time error (`NullPointerException`)

What is `this` madness?

- Code inside a class can refer to itself via the **this** keyword.
  - it only makes sense with non-static methods
  - this is the same as Python's use of **self**

```
public class Triplet {
    public int x,y,z;
    public Triplet(int x, int y, int z {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Practice Problems



Coordinate Class:

- Add a constructor method. What parameters should it have?

Square Class:

- Add a constructor method. What parameters should it have?

Knapsack Class:

- Add a different constructor method that doesn't allow for the starting `items` value. We still need to instantiate `items`.

Method Overloading (Quick View)

We can have **multiple methods with the same name** in one class! They are distinct methods with no actual relations other than the name – this is Java's approach to default parameters!

To coexist, **their method signatures must be uniquely identifiable** by the parameter types and method name alone.

- parameter names are ignored – arguments are nameless.
- return type is ignored – not explicit via method call, so can't help select correct method.

Constructors are methods too – we can write multiple constructors for one class, as long as they have different signatures.

- This is a valuable opportunity (something Python disallows).

Method Overloading - Examples

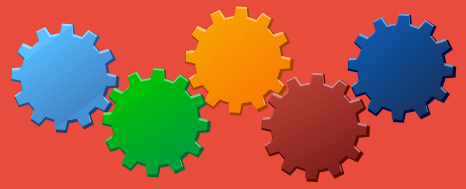
Example: these methods may all be in one class:

```
public int foo (int a, int b){…}
public int foo (char a, int b){…}
public int foo (int b, char a){…}
public int bar (int a, int b){…}
public String foo (int a, int b, int c){…}
```

The following could **not** be added to this class:

```
public String foo (int a, int b) {…}      //return type irrelevant
public int foo (int other, int names){…}  //param names irrelevant
private int foo (int a, int b){…}         //modifier irrelevant
```

Practice Problems



Method Overloading:

- Add another constructor method to the Coordinate, Square, and Knapsack classes.

  → How will it be distinguished from the other constructor method(s)?

Static methods

Same idea with static fields. If a method is declared static, all objects of that class are sharing the method. This has certain consequences:

- A static method cannot access non-static members of a class. **Rule of thumb**: a static method can't use **`this`** to refer to any instance/object of its own class.
- You don't need an object/instance of a class to call one of its static methods. You simply use ***`class_name`*`.method()`** instead of ***`object_name`*`.method()`**, e.g. **`Math.PI`**, **`Math.sqrt(2.45)`**

Hint: Static methods should always be your first choice (they're more efficient). Unless a method needs access to instance variables/methods, declare it static.

Pass-by-value *vs* pass-by-reference

- When calling a method in Java, the arguments are **passed-by-value**:

  The value of the ***argument*** is assigned to the value of the method's respective ***parameter***

- The two variables – the argument at the calling site and the local variable defined as a parameter in the method signature – live in distinct locations in memory. Modifying one doesn't modify the other.

- When the argument/parameter is not a primitive type though, but a reference type, **pass-by-value** has certain implications. Let's do some memory drawing to see what happens in this case...