
# Classes and Objects 2

Locals, globals, heap
Garbage collection
Initializers, access control
Enum types
ArrayLists

# Where Do a Program's Values Live?

## Locals (Stack)

- Declared within scope of a method (parameters, local variables)
- Temporary State
- Disappear once method returns

## Globals (Statics)

- Declared within scope of a class (static modifier)
- Global, shared
- Exist as long as class is loaded (usually, the duration of a program)

## Instances (Heap)

- Hold fields declared within scope of a class (without static modifier)
- Represent objects and their fields
- Exist as long as instance is reachable

## In the Waterloo Java Visualizer:

"Frames"

"Static fields" in "Frames"

"Objects"

# Garbage Collection

`new` creates instances/objects, who live on the heap.

What if you don't need an object anymore?

Some languages require you to manually delete them. Error-prone:

- Delete too late/never: "memory leak"
- Delete too early: references to things that do not exist anymore

Java (any many other languages) uses a garbage collector to automatically collect objects that can no longer be used. They conservatively approximate this by looking at what is reachable from any locals or globals.

# The `this` keyword

Available within instance methods and constructors – a reference to the object whose method/constructor is being executed.

Use it to

- **Disambiguate between parameters and instance field names**: when there is both a parameter `p` and an instance field `p`, `p` refers to the parameter and `this.p` to the instance field.
- **Call other constructors**: when there are multiple constructors, they may call each other using `this` as if it were a method name.

# Access Control

Access modifiers determine from where fields and methods can be accessed.

- Top level: `public` or *package-private* (no modifier)
- Member level: `public`, `protected`, *package-private*, or `private`

| Modifier | Same Class | Same Package | Subclass | World |
|---|---|---|---|---|
| `public` | ✓ | ✓ | ✓ | ✓ |
| `protected` | ✓ | ✓ | ✓ | ✗ |
| *no modifier* | ✓ | ✓ | ✗ | ✗ |
| `private` | ✓ | ✗ | ✗ | ✗ |

# More modifiers: `static` and `final`

The modifier **`static`** marks class members as opposed to instance members (without the **`static`** modifier). Reminder: class members are global – there is only one version.

The modifier **`final`** makes a variable/field unchangeable. Fields that are **`static`** ***and*** **`final`** are called ***constants*** (reminder: naming scheme is ALL_CAPS_WITH_UNDERSCORES). Instance fields and local variables can be **`final`**, too.

Classes and methods can also be **`final`**, more on that later.

# Initializers

Variables may be initialized when they are declared, as in

```
int x = 5;
```

You can also use **initializer blocks** – blocks of code enclosed by braces `{}` directly contained in the class body (static initializer blocks are preceded by the keyword `static`).

Code in

- *instance initializer blocks* runs before every constructor
- *static initializer blocks* runs when the class is first accessed

# Enum Types

An enumerated type is defined with the `enum` keyword.

A variable of an enumerated type must be one of a set of predefined values. This is useful for defining finite sets of distinct values, such as NORTH, SOUTH, EAST, WEST, or HD, D, CR, P, N, etc.

- May have other **fields**
- May have **methods**
- May use **constructors**
- Can be used as argument to **iterators**

# ArrayList

Recall: Arrays have a fixed size – a bit inflexible.

Lists can change their size.

`ArrayList<T>` is a list of items of type `T` (we'll get to generics in a bit)

Major downside: `T` can not be a primitive type.

`Person[]` vs. `ArrayList<Person>`

| Fixed-size storage for some number of Person objects | Variable-size storage for any (and changing) number of Person objects |
|---|---|

# ArrayList

Usage:

```
ArrayList<Person> people = new ArrayList<Person>();
        //new empty list, then add two people
people.add(new Person("Fred", 19));
people.add(new Person("Mary", 22));
for(Person p : people) { //iterate through list
  System.out.println(p);
}
System.out.println(p.get(1)); //print person at index 1
people.remove(0); //remove Fred, Mary now at index 0
```