Advanced Programming
Objects and Classes

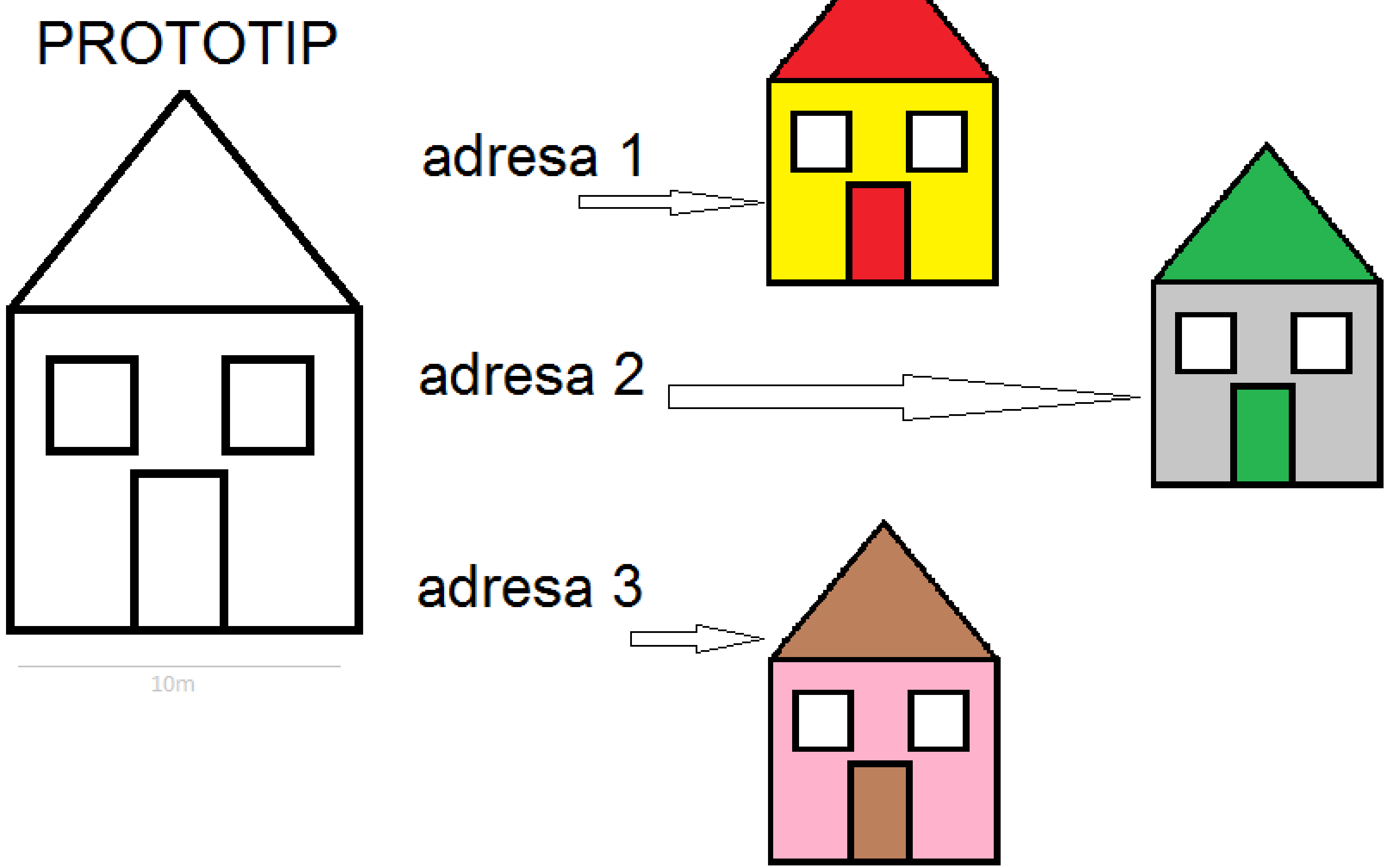
OOP Concepts

- **Object** = A software entity described by a *state* and a *behaviour*.
- **Class** = A prototype describing objects:

  an object is an instance of a class.

- **Reference** = An entity used to uniquely locate an object (may be a pointer to a memory location).
- **Program** = A dynamic set of objects interacting with each other (within the same JVM).
- **Interface** = A *contract* a class may agree to follow.
- **Package** = A *namespace* for organizing classes.
- **Module** = Runtime modularization

Class – Reference - Object

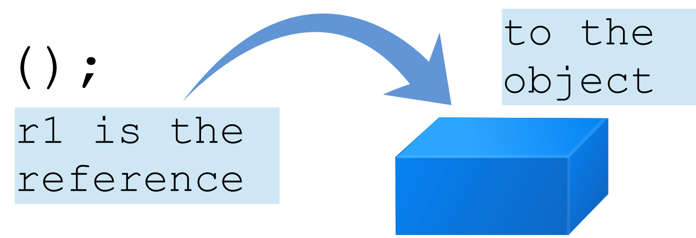


Creating Objects

<u>Declaration, Instantiation, Initialization</u>

```
ClassName refName = new ClassName([arguments]);
```

```
Rectangle r1 = new Rectangle();
```




```
Rectangle r2;
r2 = new Rectangle(0, 0, 100, 200);

Rectangle r3 = new Rectangle(
    new Point(0,0), new Dimension(100, 100));
```

NullPointerException

```
Rectangle square;
    (equivalent to: Rectangle square = null;)
square.x = 10;


Rectangle[] squares = new Rectangle[10];
squares[0].x = 10;
```

Using Objects

- **objectReference.variable**

```
Rectangle square = new Rectangle(0, 0, 100, 200);
System.out.println(square.width);
square.x = 10;
square.y = 20;
square.origin = new Point(10, 20);
```

- **objectReference.method([parameters])**

```
Rectangle square = new Rectangle(0, 0, 100, 200);
square.setLocation(10, 20);
square.setSize(200, 300);
```

Destroying Objects

Objects that are <u>not referenced</u> anymore will be automatically destroyed.

An allocated object is no longer referred when all its reference variables:

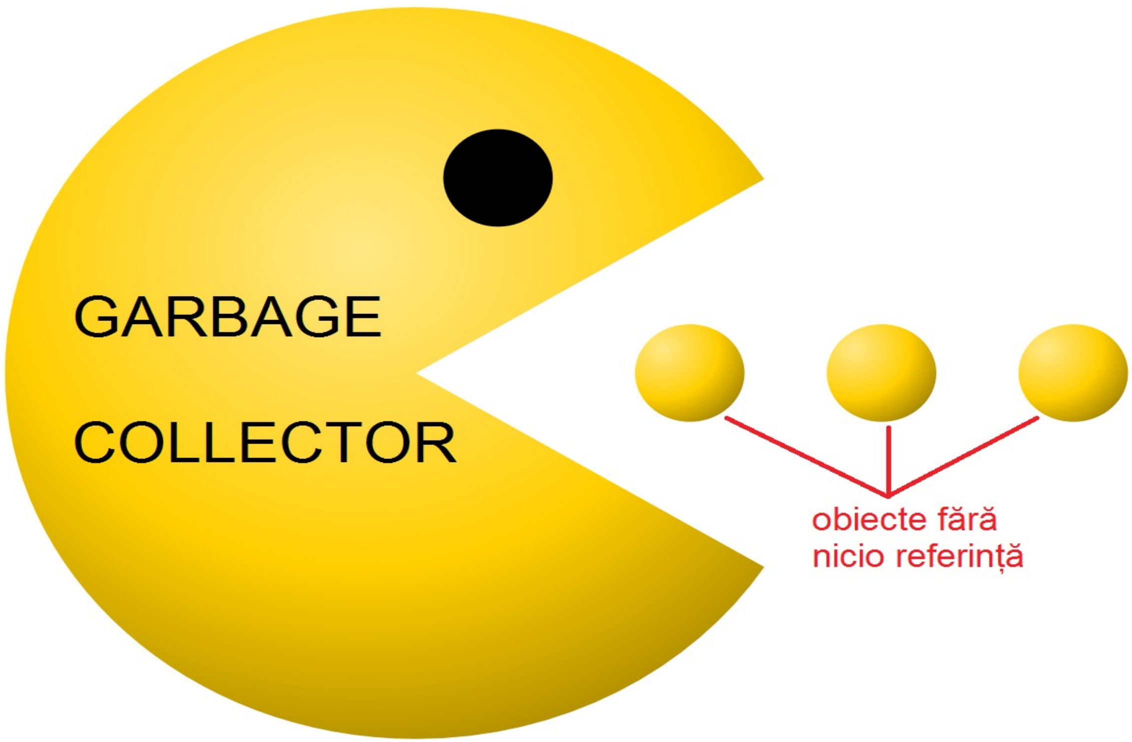
- *no longer exists (in a natural way)*
- *explicitly were set* `null`.

```
public class Test {
  String a;
  public void init() {
    a = new String("a");
    String b = new String("b");
  }
  public void stop() { a = null; }
}
```

```
Test test = new Test();
test.init();
...
test.stop();
```

Garbage Collector

A JVM component responsible with recovering memory



`System.gc()`: ”Suggests” JVM to start the Garbage Collector

The `finalize` method: invoked just before the removal of an object from memory.
@Deprecated

finalize ≠ destructor !

java ***-verbose:gc***

[GC (Allocation Failure) ***1048576K*** *→* ***31562K*** *(**4019712K**), 0.0211351 secs]*

used before GC *used after GC* *total allocated*

GC Implementations

Mark and **Sweep**

- Serial Garbage Collector
  - Stop the world
- Parallel Garbage Collector
  - Uses multiple threads for managing heap space
- CMS Garbage Collector
  - Concurrent Mark Sweep (CMS)
- **G1 Garbage Collector (default)**
  - Garbage First - partitions the heap into a set of equal-sized heap regions

Generational Collection

Memory is divided into generations, that is, separate pools holding objects of different ages. For example, the most widely-used configuration has two generations: one for young objects and one for old objects.

Generational garbage collection exploits the following observations, known as **the weak generational hypothesis:**
• Most allocated objects are not referenced (considered live) for long, that is, they die young.

• Few references from older to younger objects exist.

Young generation collections puts a premium on speed, since they are frequent, removing lots of objects that are no longer referenced.

The old generation is typically managed by an algorithm that is more space efficient.

Heap, Stack, Metaspace

- **Heap** → memory to store all the Objects.
- **Stack** → values (primitives and references) existing within the scope of the function they are created in.
- **Metaspace** → native memory for the representation of class metadata
- Adjusting memory parameters
  - *java.lang.OutOfMemoryError: -Xms1024m, -Xmx2G*
  - *java.lang.StackOverflowError: -Xss512k*
  - *XX:MetaspaceSize*
  - *java.lang.Runtime*

```
Runtime runtime = Runtime.getRuntime();
long memory = runtime.totalMemory() - runtime.freeMemory();
```

The Size of an Object

- An object allocated on the heap has a **header** which contains information used for locking, garbage collection or the identity of that object.
- The size of the header depends on the OS, and it may be **8 bytes** on 32 bit architectures or **16 bytes** on 64 bits.
- For performance reasons and in order to conform with most of the hardware architectures, JVM **align** data.

  That means that if we have an object that wraps just one byte, it will not use **8 (object header) + 1 (content) = 9** bytes of memory on the heap, but it will use 16 bytes as it needs to be aligned to the next 8 byte boundary.
- Each String object costs 24 bytes (plus its actual content). Why?

Declaring a Class

```
[public][abstract][final]class ClassName
     [extends SuperclassName]
     [implements Interface1 [, .. ]] {

     The Class Body
          Variables
          Constructors
          Methods
          Nested classes
}
```

Example

Single-responsibility principle

```
public class Person {

    private int id;
    protected String name;

    public Person() { }

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

SOLID

Afraid of Creating Objects?

```
public class Main {

  public static void main(String args[]) {
    int nbObjects = 1_000_000;
    int nbSteps = 1_000;
    Main app = new Main();
    for (int k = 0; k < nbSteps; k++) {
      app.testObjects(nbObjects);

  }
  private void testObjects(int n) {
    long t0 = System.currentTimeMillis();
    Person[] persons = new Person[n];
    for (int i = 0; i < n; i++) {
      persons[i] = new Person();
    }
    for (int i = 0; i < n; i++) {
      persons[i].setName("Person " + i);
    }
    long t1 = System.currentTimeMillis();
    System.out.println(t1 - t0);
  }
}
```

```
Using 1000000 objects: 69 ms
Using 1000000 objects: 57 ms
Using 1000000 objects: 54 ms
Using 1000000 objects: 62 ms
Using 1000000 objects: 93 ms
Using 1000000 objects: 94 ms
Using 1000000 objects: 83 ms
Using 1000000 objects: 86 ms
Using 1000000 objects: 59 ms
Using 1000000 objects: 56 ms
[GC (Allocation Failure)
1048576K->31610K(4019712K),
0.0230124 secs]
Using 1000000 objects: 81 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 33 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 31 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 33 ms
Using 1000000 objects: 32 ms
Using 1000000 objects: 32 ms
[GC (Allocation Failure)
1080186K->14514K(4019712K),
0.0120782 secs]
Using 1000000 objects: 47 ms
...
```

Access Level Modifiers

Controlling Access to Members of a Class

Modifier	Class	Package	Subclass	World
public	✔	✔	✔	✔
protected	✔	✔	✔	⛔
no modifer	✔	✔	⛔	⛔
private	✔	⛔	⛔	⛔

Inheritance

Single inheritance

A class has *one and only one* direct superclas

… except of?

```
public class Student extends Person {
   // Person is the superclass of Student
   // Student is a subclass of Person
}
```

No multiple inheritance of implementation

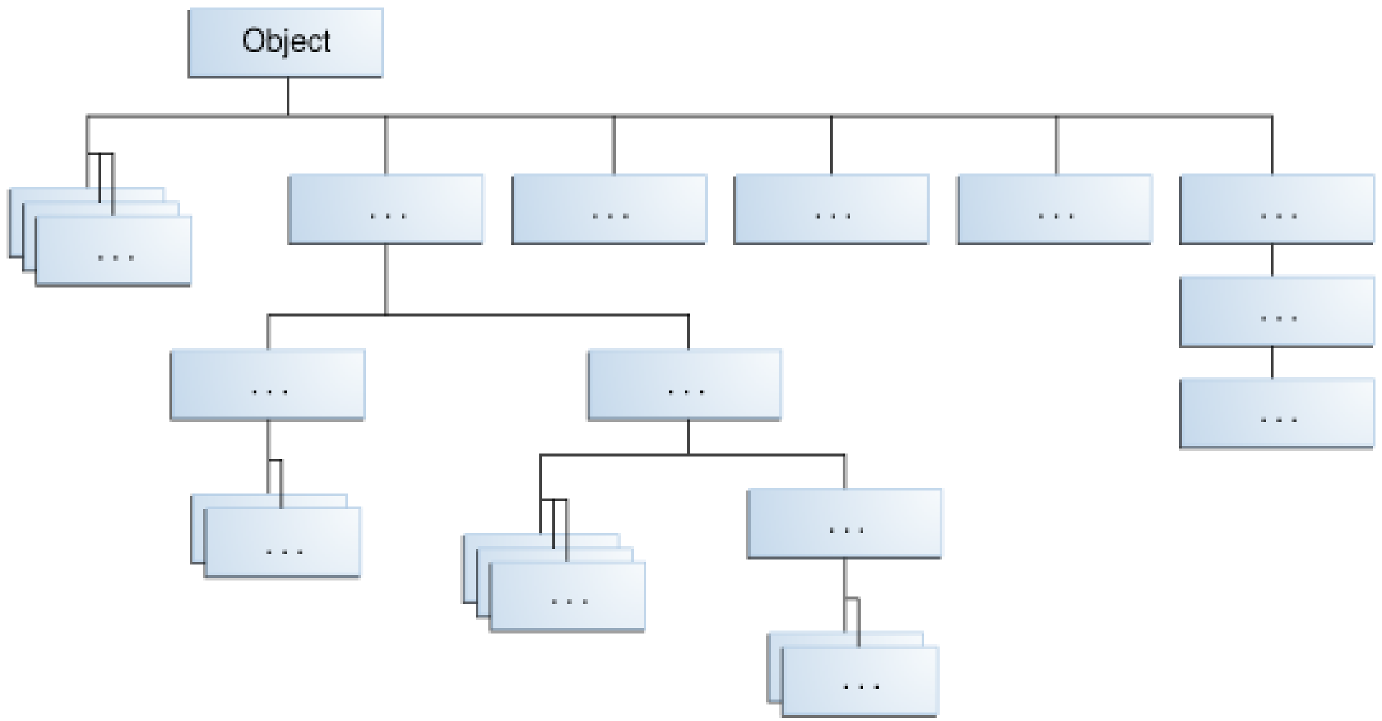
```
public class Student extends Person, Robot {
   // Syntax Error
}
```

The *Object* Class

Object is the root of the class hierarchy.

Every class has *Object* as a superclass.
All objects, including arrays, implement the methods of this class.

```
class A {}
class A extends Object {}
```



Object Class Methods

All objects, including arrays, implement the methods of the *Object* class:

- **`toString`** : Returns a string representation of the object.
- **`equals`** : Indicates whether some other object is *"equal to"* this one.
- **`hashCode`** : Returns a *hash code* value for the object.
- **`getClass`** : Returns the runtime class of this object.
- **`clone`** : Creates and returns a copy of this object (by default, a *shallow copy*)
- **`finalize`** : Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- `...`

Example
Overriding *Object* Methods

```
public class Complex {
  private double a, b;
  public Complex add(Complex comp) {
    return new Complex(a + comp.a, b + comp.b);
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof Complex)) return false;
    Complex comp = (Complex) obj;
    return ( comp.a==a && comp.b==b);
  }
  @Override
  public String toString() {
    String semn = (b > 0 ? "+" : "-");
    return a + semn + b + "i";
  }
}
...
Complex c1 = new Complex(1,2); Complex c2 = new Complex(2,3);
System.out.println(c1.add(c2));      // 3.0 + 5.0i
System.out.println(c1.equals(c2));   // false
```

Object.hashCode()

- A **hash function** takes as input some data of arbitrary size and maps it to a value of a fixed length (called hash value).
- The **hashCode** method takes as input an object (this) and returns a hash value for that object. This method is supported for the benefit of **hash tables**.
- If two objects are equal according to the **equals** method, they must produce the same integer hashCode.
- It is **not required** that if two objects are unequal, they must produce distinct integer results.
- As much as is reasonably practical, the default hashCode method defined by **Object** does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer)

Example String.hashCode()

```
h(s)=s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
(s[i] is the i-th character of the string)
```

```
public int hashCode() {
   int h = hash; //hash is by default 0
   if (h == 0 && value.length > 0) {
       char val[] = value;
       for (int i = 0; i < value.length; i++) {
          h = 31 * h + val[i];
       }
       hash = h;
     }
     return h;
}
```

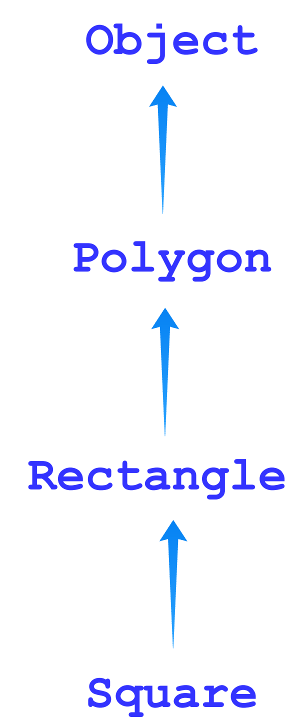
Reference Type – Class Instance

An object can be reffered by a variabile with a proper type.
Liskov substitution principle (LSP): if S is a subtype of T, then objects of type T may be replaced with objects of type S.

```
Square ref1     = new Square();
Rectangle ref2 = new Square();
Polygon ref3    = new Square();
Object ref4     = new Square();
```

~~Square badRef = new Rectangle();~~

```
Polygon metoda1( ) {
  if (…)
    return new Square();         // Correct
  else
    return new Rectangle();     // Correct
}
Rectangle metoda2( ) {
  if (…)
    return new Polygon();        // Error
  else
    return new Square(;          // Correct
}
```



SOLID

Class Constructors

A constructor has the same name as its class, has no explicit return type, cannot be abstract, static, final, or synchronized.

```
public class ClassName {
  [modifiers] ClassName([arguments]) {
    // Constructor
  }
}
```

```
class A {
    protected int x;
    public A(int x) { this.x = x; }
    public A() { this(0);}
}

class B extends A{
    public B(int x) { super(x);}
}

class C {
    //Default (implicit) constructor
    //Generated by the compiler (if necessary)
}
```

Invoking Constructors

```
class A {
    public A() {
        System.out.println("A");
    }
}

class B extends A {
    public B() {
        System.out.println("B");
    }
}

class C extends B {
    public C() {
        System.out.println("C");
    }
}
```

```
C c = new C();
```

Class Methods

```
public class ClassName {
  [modifiers] ReturnedType methodName([arguments]) {
    // The body of the method
  }
}
```

```
class A {
    public void hello() {
        System.out.println("Hello");
    }
    public void hello(String str) {
        System.out.println("Hello " + str);
    }
}
class B extends A {
   @Override
    public void hello() {
        super.hello();
        System.out.println("Salut");
    }
    @Override
    public void hello(String str) {
        System.out.println("Salut " + str);
    }
}
```

Overloading

Overriding

Sending Parameters

Always pass-by-value!

```java
void method(StringBuilder s1, StringBuilder s2, int number)
{
  // StringBuilder is a reference data type
  // int is a primitive data type
  s1.append("bc");
  s2 = new StringBuilder("yz");
  number = 123;
}
...
StringBuilder s1 = new StringBuilder("a");
StringBuilder s2 = new StringBuilder("x");
int n = 0;
method(s1, s2, n);
System.out.println(s1 + ", " + s2 + ", " + n);
```

Variable Number of Arguments

```
[modifiers] ReturnedType methodName(ArgumentsType ... args)
```

```
void method(Object ... args) {
  for(int i=0; i<args.length; i++) {
    System.out.println(args[i]);
  }
}
...
method("Hello");
method("Hello", "Java", 1.8);
```

```
System.out.printf("%s %d %n", "GrandTotal:", 1000);
```

The *final* Modifier

"Controlling" the Open-Closed principle

- **Final Variables** – once initialized, cannot be modified

```
final int MAX = 100; . . . ~~MAX = 200;~~

final int n; . . . n = 100; . . . ~~n = 200;~~
```

In constructors!

- **Final Methods** – cannot be overridden

```
class Student {
  ...
 final float getGrade(float scores[])
  {
    ...
  }
}
```

```
class ComputerScienceStudent
    extends Student {
  ~~float getGrade(float scores[])~~ {
    return 10.00;
  }
}// Compile error!
```

- **Final Classes** – cannot be extended

```
final class A{}, ~~class B extends A {}~~
```

SOLID

The *static* Modifier

Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

- **Static variables** – hold values specific to a certain class and not for every instance.

  Example: efficient declaration of constants

  ```
  static final double PI = 3.14;
  ```

- **Static methods** – available at the class level and not for every instance (can only access static variables)

  Example: “global” methods

  ```
  double x = Math.sqrt(2);
  ```

Example: Using Static Members

```
public class Dog {
    private String name;
    public static final String MESSAGE = "Come here, ";
    public static final String DEFAULT_BARK = "Woof!";

    public Dog(String name) {
        this.name = name;
    }
    public void come() {
        System.out.println(MESSAGE + name + "!");
    }

    public static void bark() {
        System.out.println(DEFAULT_BARK);
        if (name.equals("Peanut")) System.out.println("Yip!");
        //non-static variable name cannot be referenced from a static context
    }

    public static void main(String args[]) {
        Dog wolfy = new Dog("Wolfy");
        wolfy.come();
        wolfy.bark(); //warning: accesing static method (or field)
        System.out.println("Wolfy barks like this: " + wolfy.DEFAULT_BARK);

        Dog.bark();
        System.out.println("A dog barks like this: " + Dog.DEFAULT_BARK);

        Dog.come();
        //non-static method come() cannot be referenced from a static context
    }
}
```

Static Initializer Blocks

Class-Level “Constructors”

```
static {
  // Initializer Block
  /*A block of code that runs only
   one time, and it is run before
   any usage of that class*/
}
```

```
public class Test {
  static int x = 0, y, z;

  // Static initializer block
  static {
    System.out.println("Initializing Class...");
    int t=1;
    y = 2;
    z = x + y + t;
  }
  public Test() { ... }
}
```

Nested Classes

Classes declared within other classes

```
public class OuterClass {

  static class StaticNestedClass {
    // No difference to any other class
    // May be private, protected, public
  }
  private class InnerClass1 {
    // Member Class
    // Access to all members of the outer class
  }
  void method() {
    class InnerClass2 {
      // Local Class (similar to closures)
      // Acces to all members of the outer class
      // and only to the final variabiels of the method
    }
  }
}
```

Compiling nested classes

`OuterClass.class,`
`OuterClass$InnerClass1.class, OuterClass$InnerClass2.class`

Abstract Classes and Methods

```
[public] abstract class AbstractClass {
   // Abstract Methods (no implementation)
   abstract ReturnedType abstractMethod([args]);

   // Normal Methods
  ...
}
```

- An abstract class defines a **template** on which concrete classes can be created (by subclassing them)
- Used to share code among several closely related classes.
- Cannot be instantiated.

Examples:

`java.awt.Component``: Button, List, ...`

`java.lang.Number``: Integer, Double, ...`

Boxing and Unboxing

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

```
Integer refi = new Integer(1);
int i = refi.intValue();

Boolean refb = new Boolean(true);
boolean b = refb.booleanValue();
```

```
Integer refi = 1; //(auto)boxing
int i = refi;     //(auto)unboxing

Boolean refb = true;
boolean b = refb;
```

Enum Types

```
public enum Signal {
    RED, YELLOW, GREEN;
}
```

```
public class TrafficLights {
    Signal signal;
    public TrafficLights(Signal signal) {
        this.signal = signal;
    }
    public boolean isCrossingAllowed() {
        switch (signal) {
            case Signal.GREEN: return true;
            default: return false;
        }
    }
}
    new TrafficLights(Signal.YELLOW).isCrossingAllowed();
```

Enums are transformed by the compiler into classes; they contain some other methods : **`Signal.values()`**

Creational Design Patters

You may want to learn about:

- Singleton
- Object Factory
- Object Pool
- Prototype
- Builder
- ...