

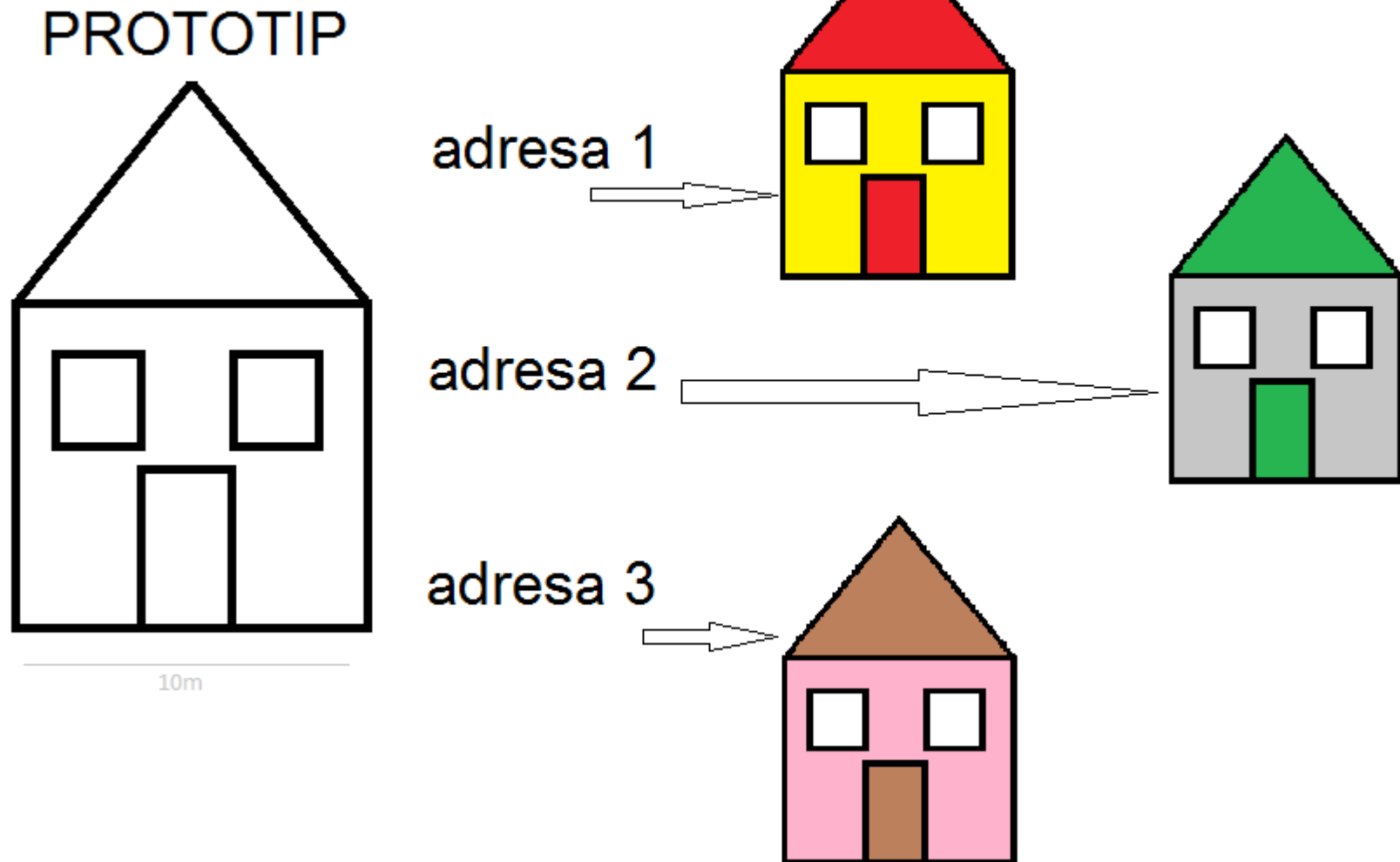


# Advanced Programming Objects and Classes

# 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.

# Class – Reference - Object



# Creating Objects

## Declaration, Instantiation, Initialization

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

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

r1 is the  
reference

to the  
object



```
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 not referenced anymore will be automatically destroyed.

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

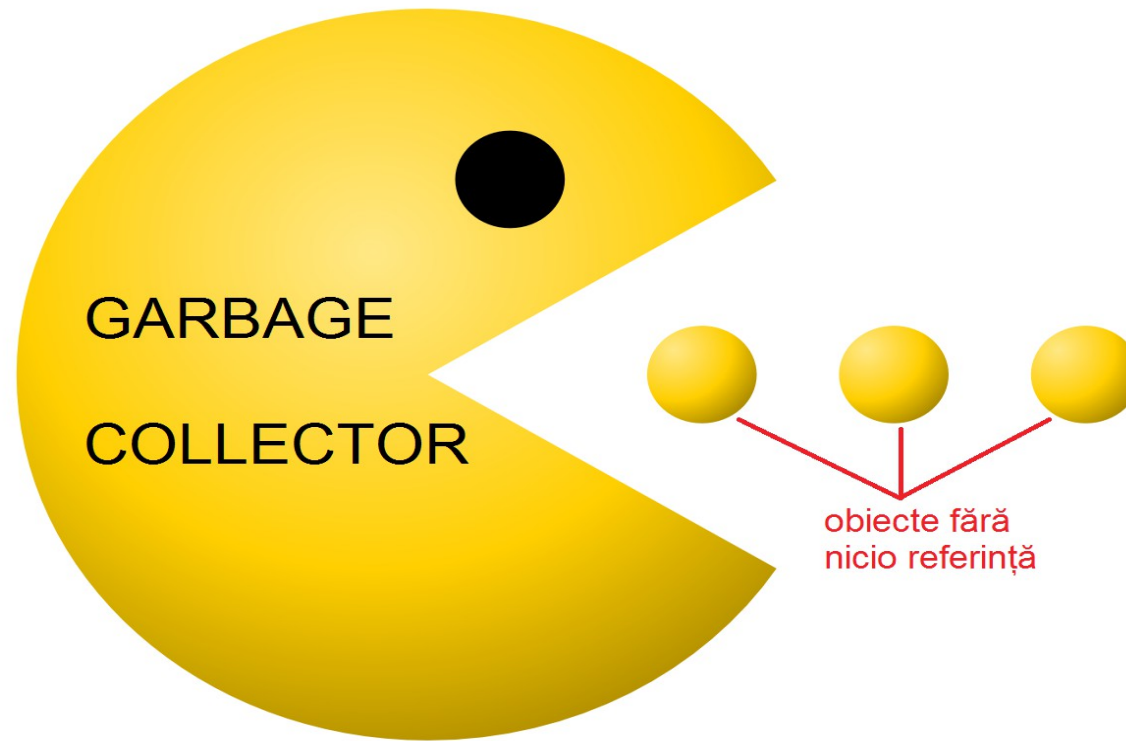
- *no longer exists (in a natural way)*
- *explicitly were set* `null`.

```
public class Test {  
    String a;  
    public void init() {  
        a = new String("aa");  
        String b = new String("bb");  
    }  
    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.  
finalize  $\neq$  destructor !

**java -verbose:gc**

[GC (Allocation Failure) **1048576K** → **31562K** (**4019712K**), 0.0211351 secs]  
used before GC used after GC total allocated



# 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();
```

# Declaring a Class

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

## The Class Body

*Variables*

*Constructors*

*Methods*

*Nested classes*

```
}
```

# Example

```
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;  
    }  
}
```

















# 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
<b>public</b>				
<b>protected</b>				
<i>no modifier</i>				
<b>private</b>				

# Inheritance

## Single inheritance

A class has *one and only one* direct superclass  
... 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

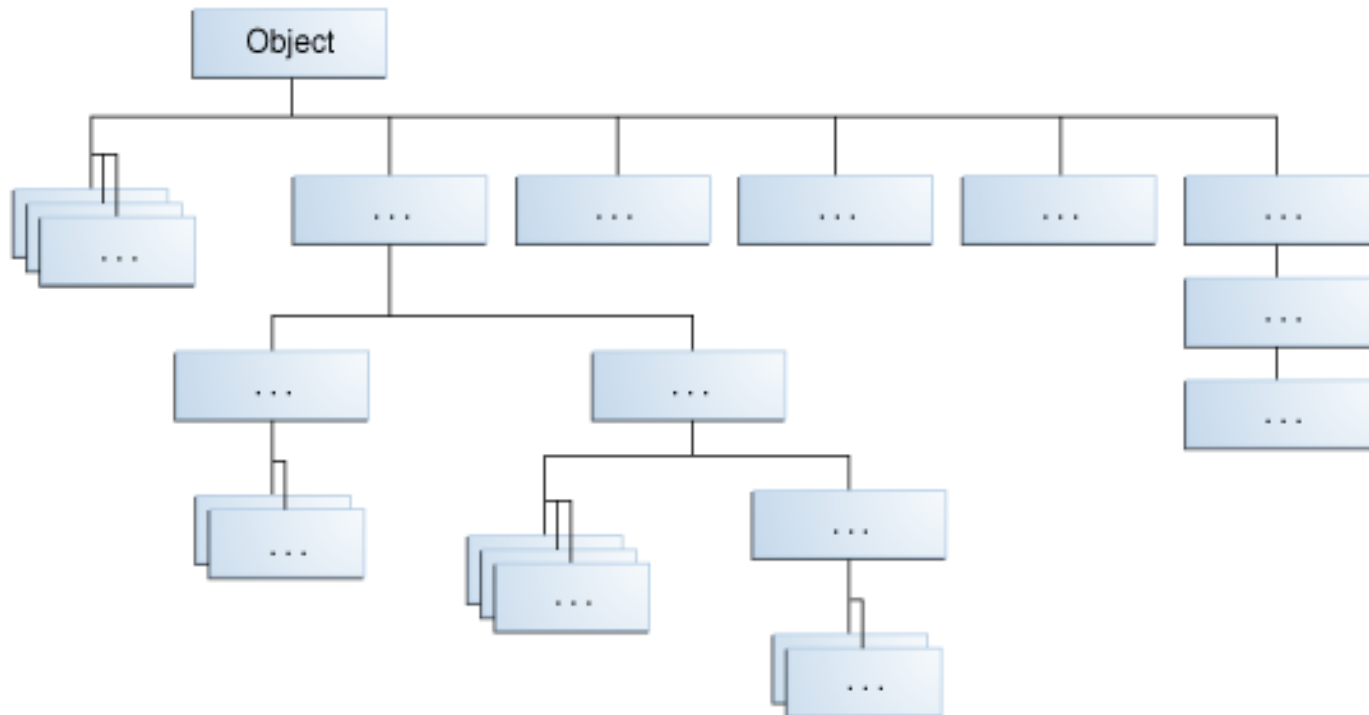
*Object* is the root of the class hierarchy.

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
```

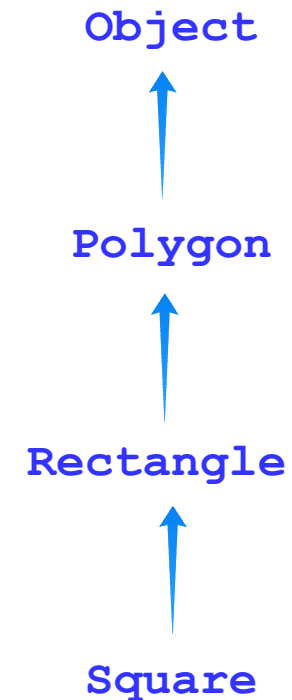
# Reference Type – Class Instance

An object can be reffered by a variable with a **proper** type.

```
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  
}
```



# Class Constructors

```
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);  
    }  
}
```

**Overloading**

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

**Overriding**

# Sending Parameters

Always pass-by-value!

```
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

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

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

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

- **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 {}
```

# 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 Example {  
    int x ;           // Instance-level variable  
    static long n;    // Class-level variable  
    public void instanceMethod() {  
        n ++; // Correct  
        x --; // Correct  
    }  
    public static void staticMethod() {  
        n ++; // Correct  
        x --; // Compile error!  
    }  
}  
  
Example.staticMethod();           // Correct  
Example obj = new Example();  
obj.staticMethod();               // Correct, Not recommended  
  
Example.instanceMethod();       // Error  
Example obj = new Example();  
obj.instanceMethod();           // Corect
```

# 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

Classed declared within other classes

```
public class OuterClass {  
    private class InnerClass1 {  
        // Member Class  
        // Access to all members of the outer class  
    }  
    static class StaticNestedClass { ... }  
  
    void method() {  
        class InnerClass2 {  
            // Local Class  
            // 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 Patterns

You may want to learn about:

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