Object Oriented Programming

LESSON 06

Inheritance and Polymorphism (Part 1)

Outline

1. What is Upcasting?
2. Dynamic link
3. What is Polymorphism?
4. Downcasting
5. Abstract class

Overview

In this chapter, you are going to learn about

- Know upcasting and how to use
- Know how dynamic links works
- Know polymorphism and its definition
- Know downcasting and how to use
- Know class abstract, its advantages and how to use it

Learning content

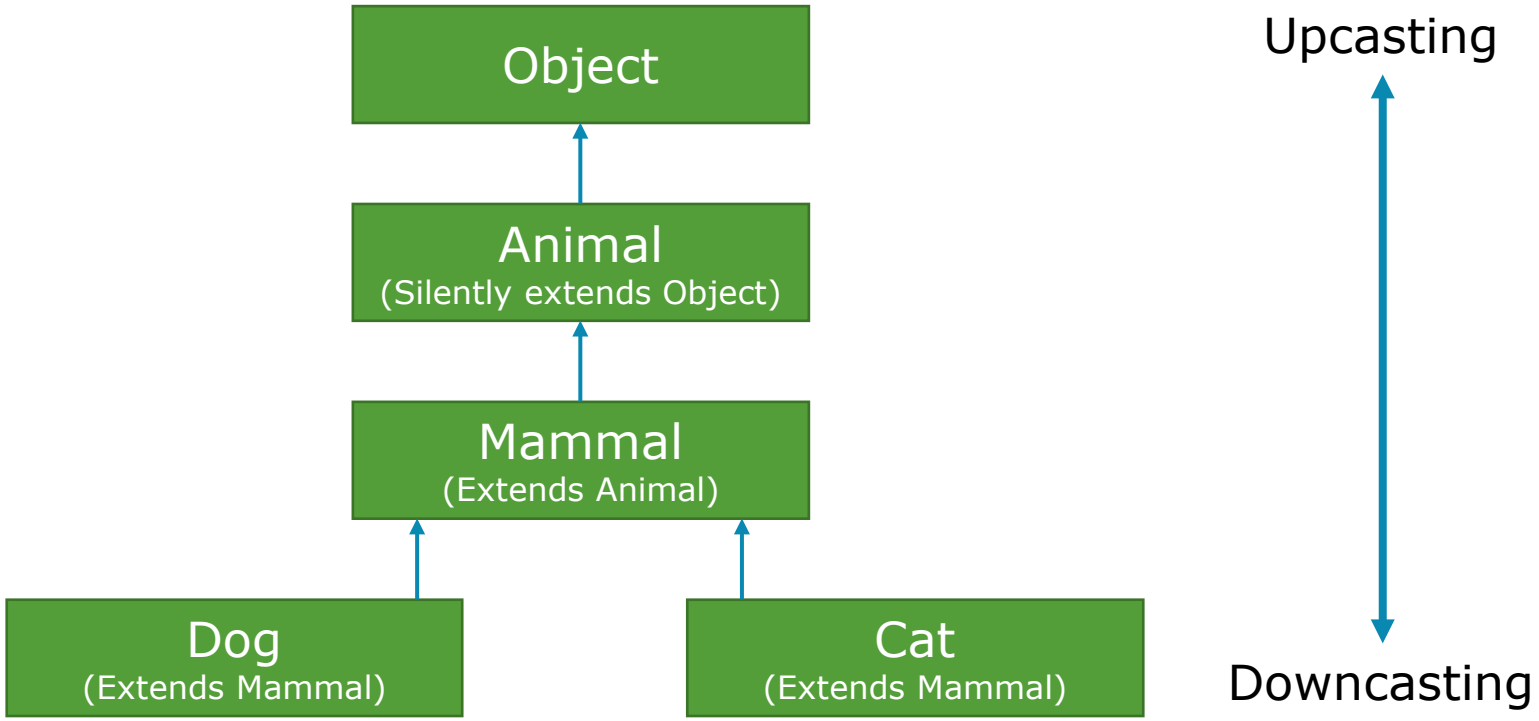
1. What is Upcasting?
   - Sub-element
   - Upcasting in Java
   - Upcasting Example
2. Dynamic Link
   - Dynamic binding
   - Static binding
   - Choice of methods, code selection
3. What is Polymorphism?
   - Usage and Advantage of Polymorphism
   - Overloading and Polymorphism
   - About method equals()
4. Downcasting
   - Force to a kind
   - Compatible class
   - Example
5. Abstract Class
   - Abstract class example
   - Requirement to use Abstract Classes
   - Better structure of above Example

Pre-Test

Question	Possible answers	Correct Answer	Question Feedback
Which 2 objects are in the same class?	a) Rat b) Mouse c) Worm	a) Rat b) Mouse	Worm has no legs, but Rat and Mouse have both 4 legs.
Which one is not polygon?	a) Triangle b) Ellipse c) Rectangle	b) Ellipse	Ellipse is not polygon, it has no vertex, it is an oval
Which one of these shapes that its surface can be calculated using one of another shape's algorithm?	a) Ellipse b) Triangle c) Rectangle	b) Triangle c) Rectangle	We can calculate surface of Rectangle using method calculate surface of Triangle.

1. Upcasting

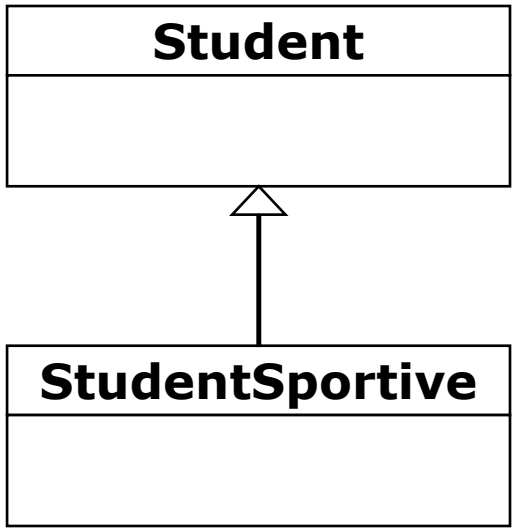
- Java permits an object of a subclass type to be treated as an object of any superclass type.



1.1. Sub-element

- Code reuse is an important aspect of inheritance, but it is perhaps not the most important
- The second key point is the relationship that links a class to its superclass:

A class B inherits from class A can be viewed as a subtype (subset) of the type defined by the class A.

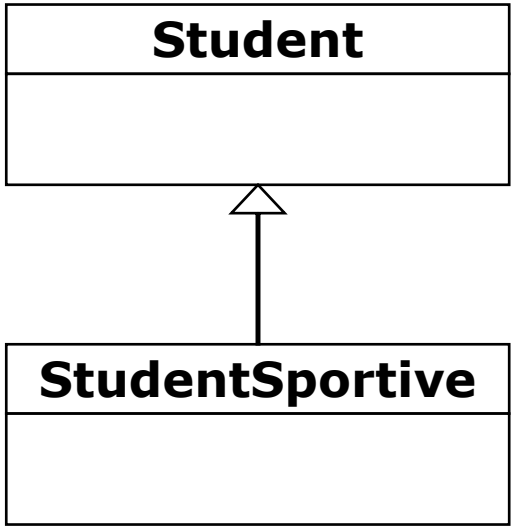


- A **StudentSportive** is a **Student**
- All sports students is included in all students

1.2. Upcasting in Java

- any instance object of the class B can also be seen as an instance of class A.
- This relationship is directly supported by JAVA:
  - to a declared reference type A it is possible to assign a value that is a reference to an object of type B (or upgrade upcasting)

```
Student s;
s = new StudentSportive(…);
```

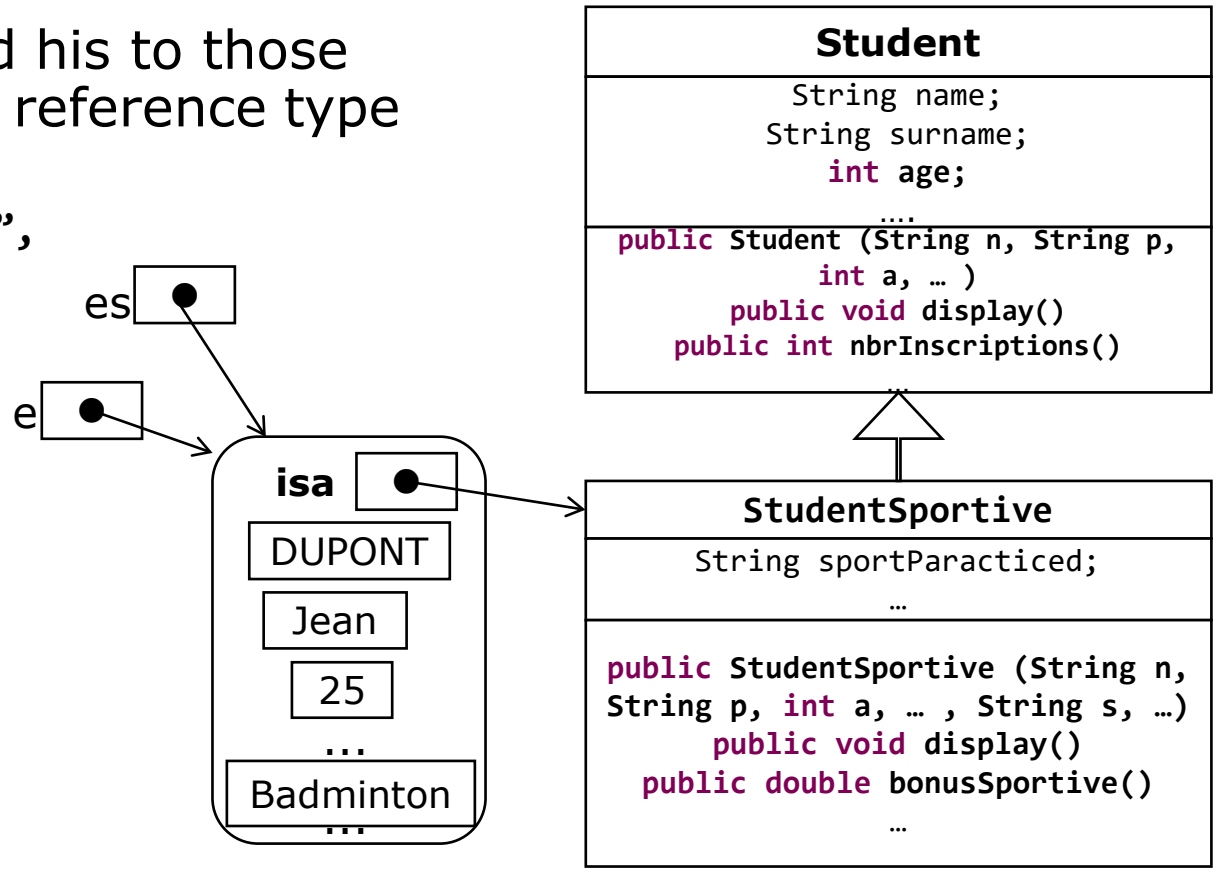


- *more generally to a reference of a given type, it is possible to assign a value that corresponds to a reference to an object whose actual type is any direct or indirect subclass of the type of reference*

1.3. Upcasting example

- When an object is "upgraded" it is seen as an object the type of the reference used to describe
  - the features are so restricted his to those proposed by the class of the reference type

```
StudentSpotive es;
es = new StudentSportive(“DUPONT”, “Jean”,
                         25,.., “Badminton”,..);
Student e;
e = es;    // upcasting

e.display();
es.display();

e.nbrInscriptions();
es.nbrInscriptions();

es.bonusSpotive():
e.bonusSportive();
```

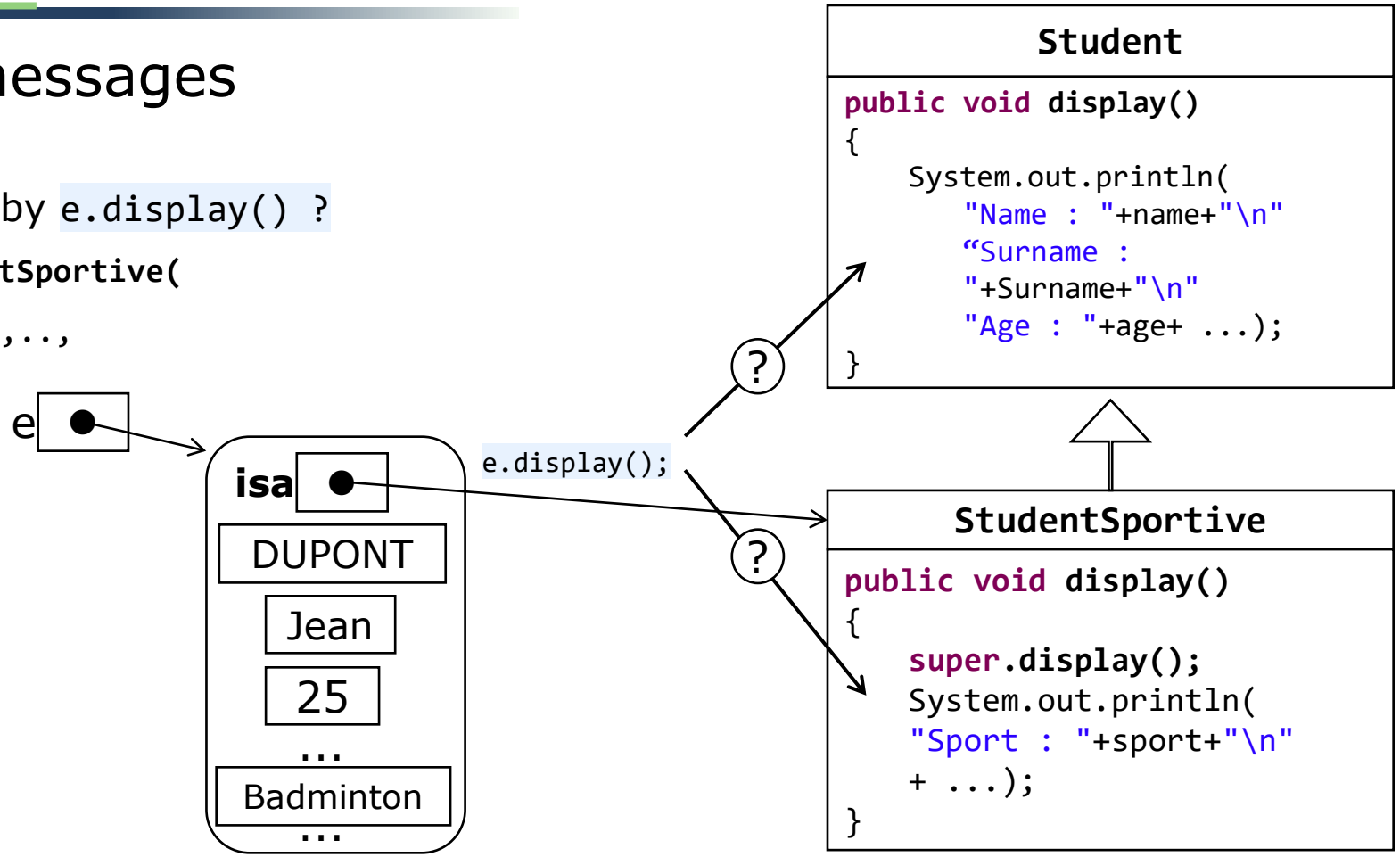


2. Dynamic Link

- Resolution messages

- What will be given by `e.display()` ?

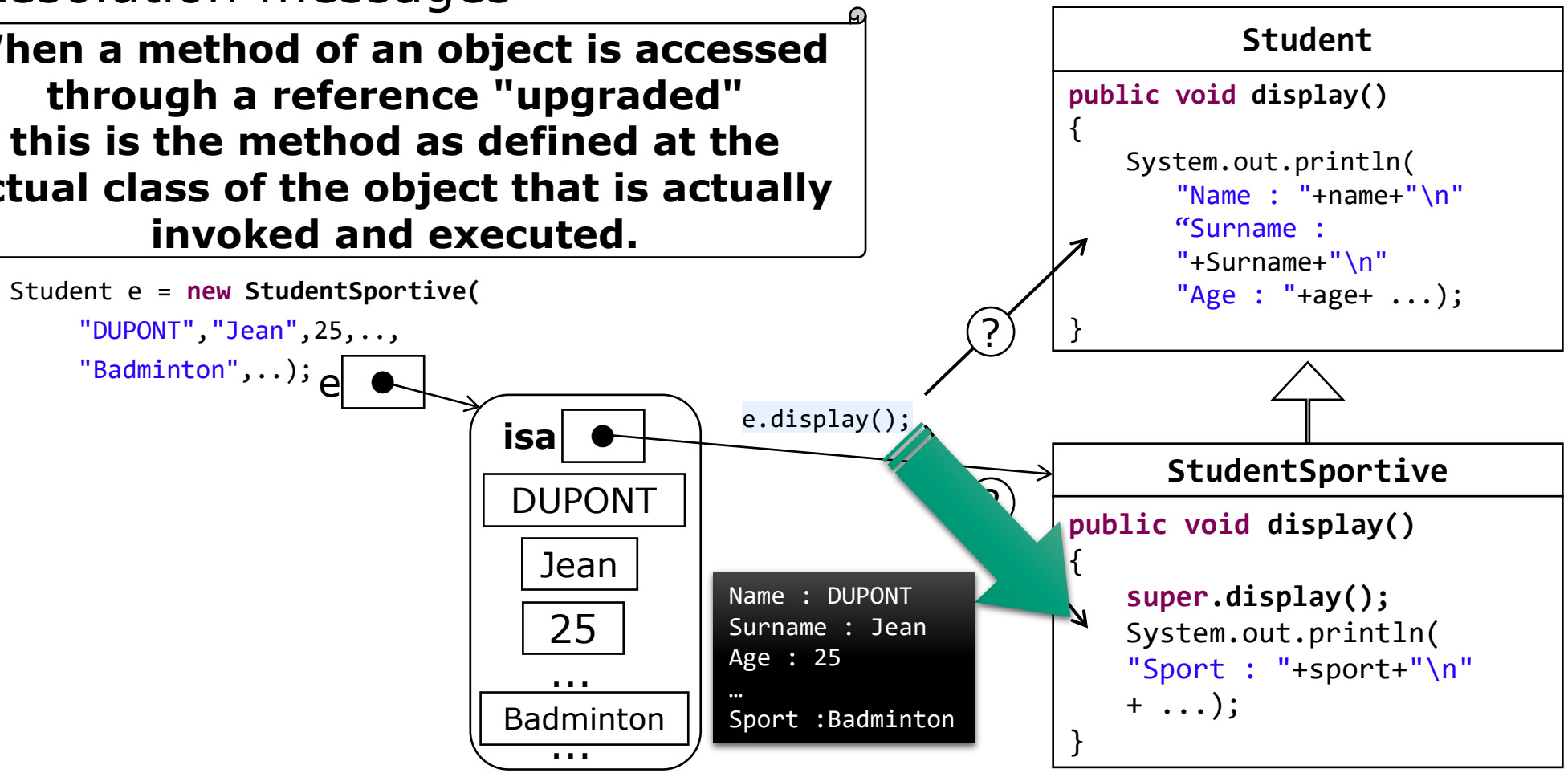
```
Student e = new StudentSportive(
    "DUPONT","Jean",25,..,
    "Badminton",..);
```



2. Dynamic Link

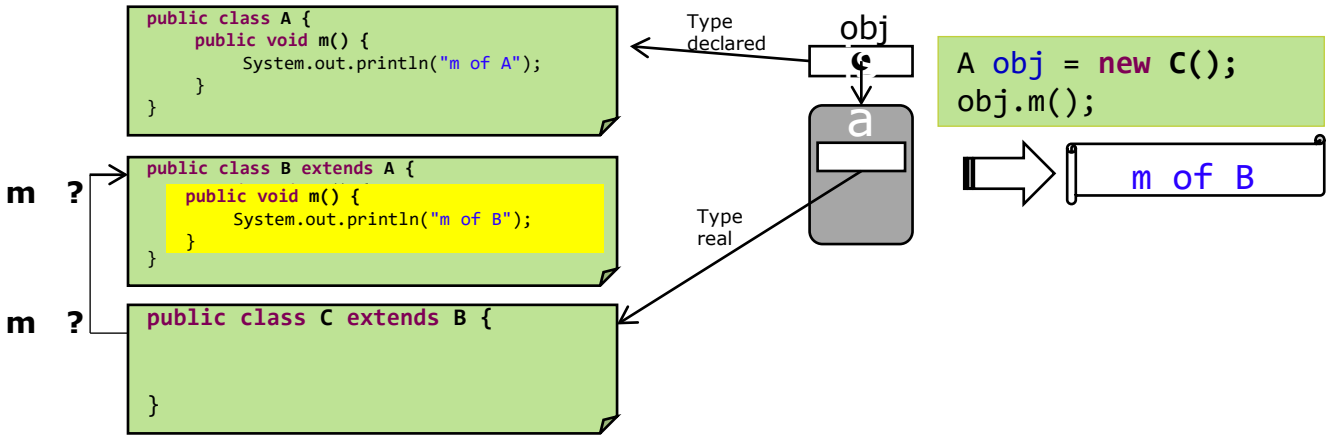
- Resolution messages

When a method of an object is accessed through a reference "upgraded" this is the method as defined at the actual class of the object that is actually invoked and executed.



2.1. Dynamic binding

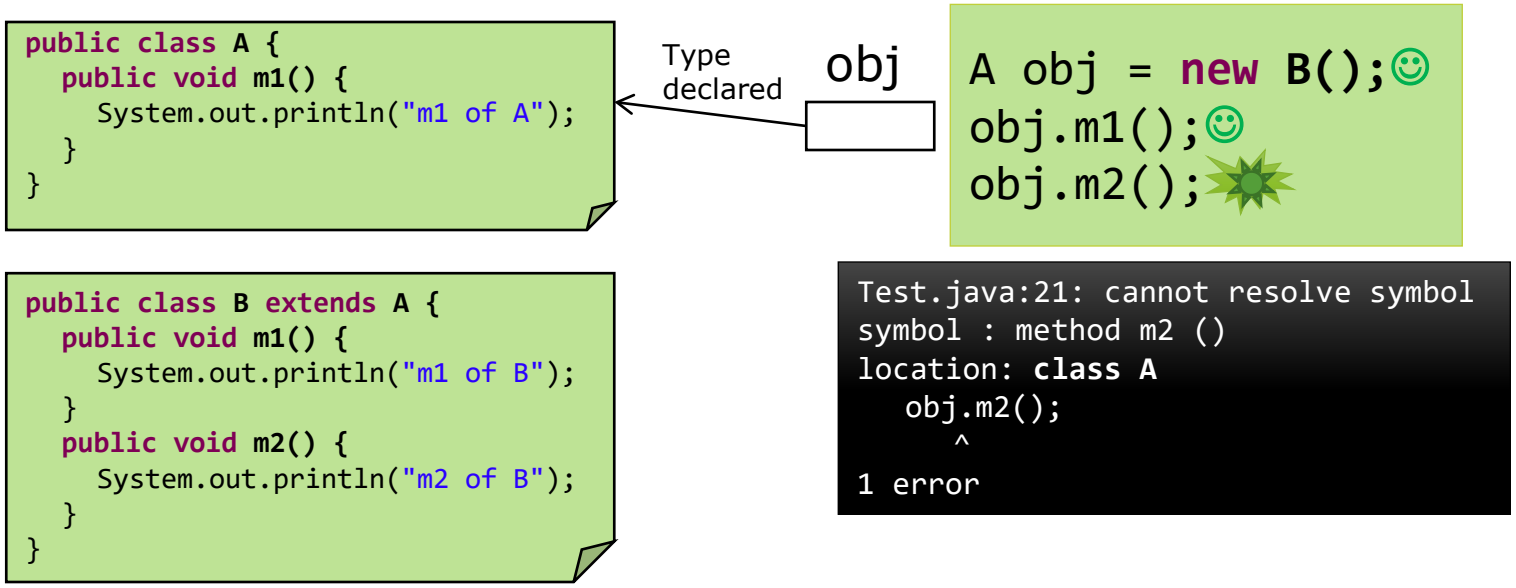
- Messages are resolved at runtime.
  - the executed method is determined at runtime (run-time) and not at compile time.
  - at that time the exact type of the object receiving the message is known.
    - the method defined for the actual type of the object receiving the message is called (not defined for its declared type).



- This mechanism is referred to as dynamic-link (dynamic binding, late binding or run-time binding)

2.2. Statistic binding

- A compilation: only static checks that are based on the declared type of the object (the reference) are performed? the declared class of the object receiving the message must have a method whose signature matches the called method.

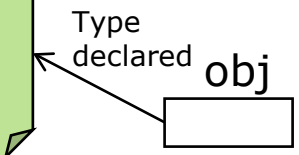


2.2. Statistic binding

- to compile it is not possible to determine the exact type of the object receiving a message

```
public class A {
   public void m1() {
      System.out.println("m1 of A");
   }
}
```

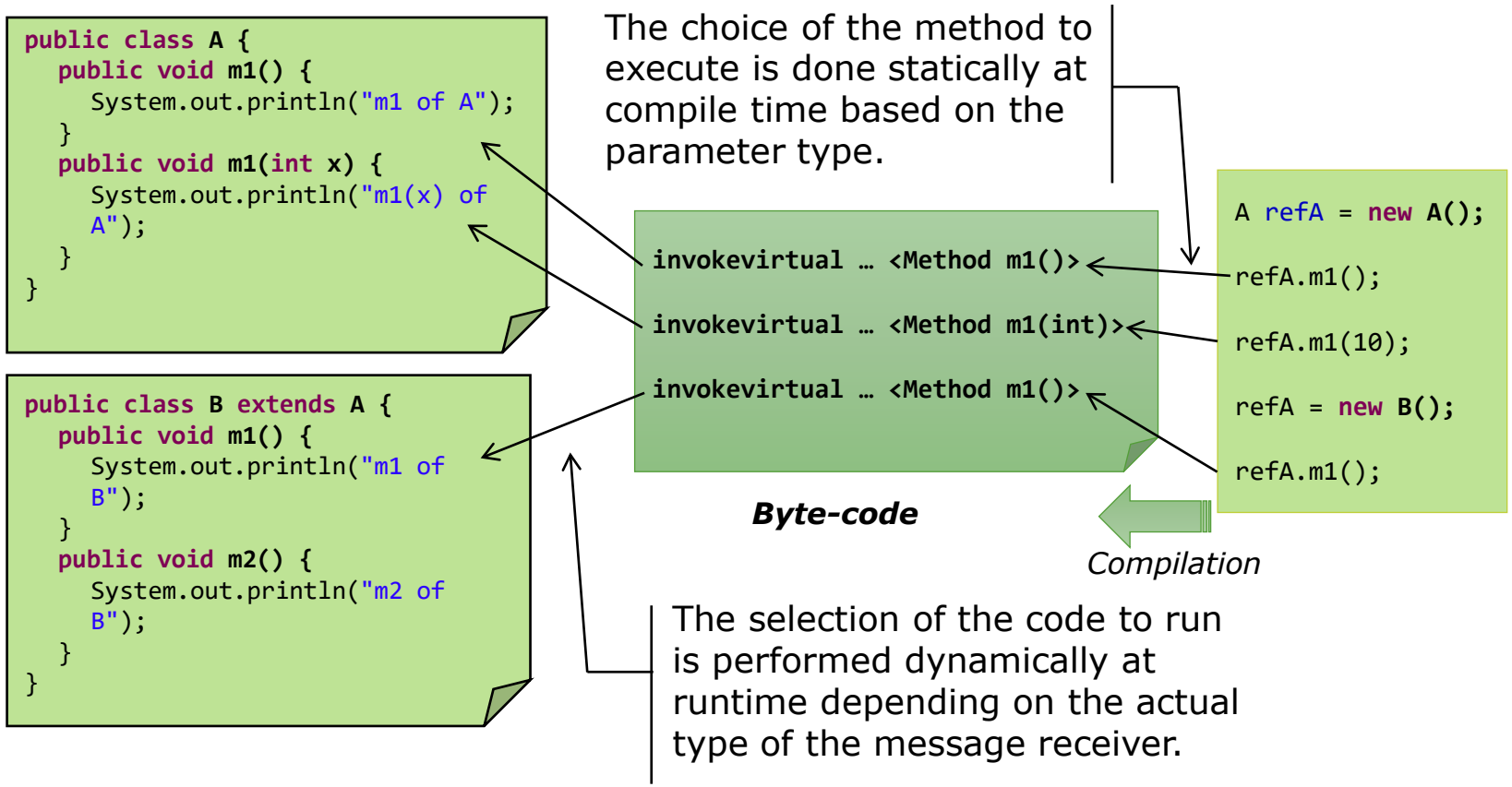
```
public class B extends A {
   public void m1() {
      System.out.println("m1 of B");
   }
   public void m2() {
      System.out.println("m2 of B");
   }
}
```




```
A obj;
for (int i = 0; i < 10; i++)
{
     hasard = Math.random()
     if ( hasard < 0.5)
        obj = new A();
     else
        obj = new B();

     obj.m1();
}
```

- Static verification: guarantees at compile time that messages can be resolved at runtime

2.3. Choice of methods, code selection



3. Default Constructor

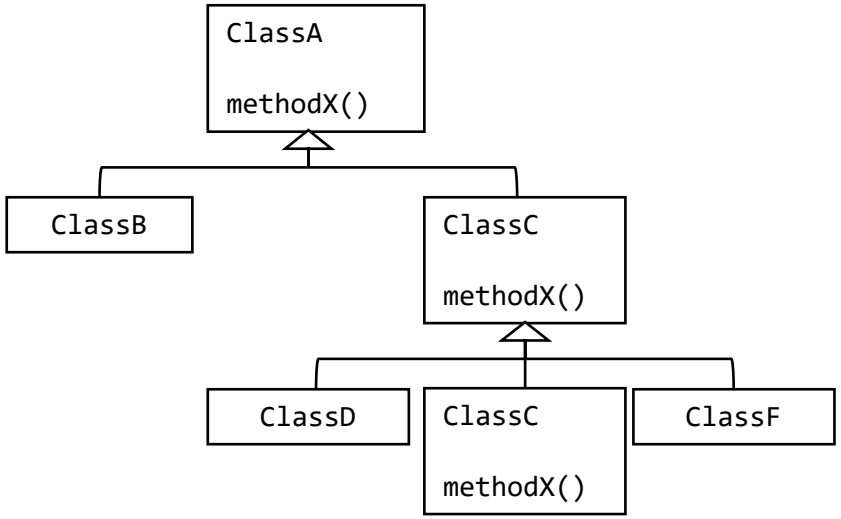
- What are the upcasting and Dynamic Link?
- In the implementation of the polymorphism

- The term polymorphism describes the characteristic of an element that can take many forms, such as water which is solid, liquid or gaseous.

- En programmation Objet, on appelle polymorphisme
  - le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.
  - le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.
- "Polymorphism is the third essential feature of an object oriented language after data abstraction (encapsulation) and inheritance"

  Bruce Eckel "Thinking in Java"

3.1. Usage and Advantage of Polymorphism




```
ClassA objA;

objA = …

objA.methodX();
```

Upcasting
reference may designate different class of objects (in any class ClassA)

+

Lien dynamique
Le comportement est différent selon la classe effective de l'objet

a special case of polymorphism.

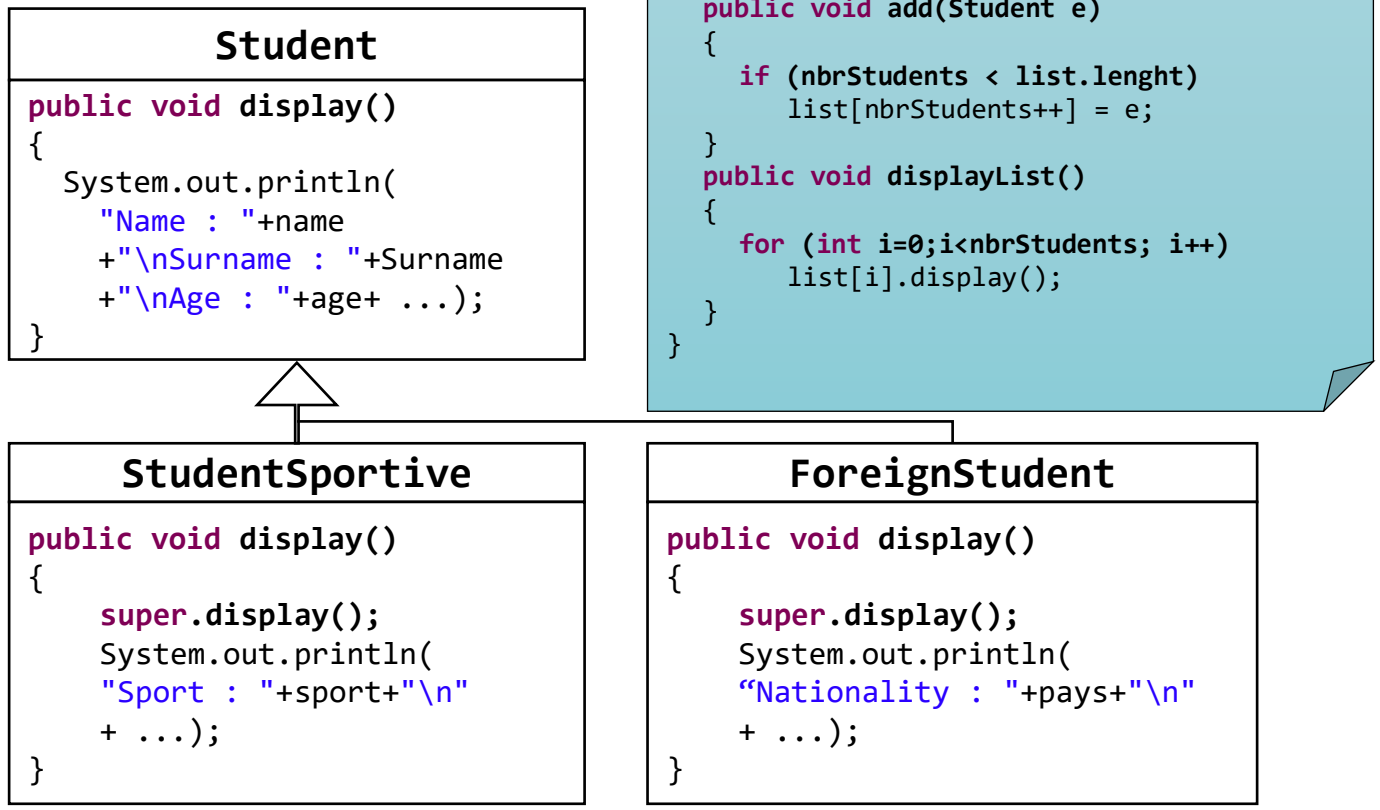
uniform handling of objects of several classes via a common base class

3.1. Usage and Advantage of Polymorphism

- `list` may contain students of any type

```
GroupTD td1 = new GroupTD();
td1.add(new Student("DUPONT", …));
td1.add(new StudentSportive("BIDULE",
"Louis", … , "ski alpin");
```

```
public class GroupTD{
   Student[] list = new Student[30];
   int nbrStudents = 0;
   ...
   public void add(Student e)
   {
      if (nbrStudents < list.lenght)
         list[nbrStudents++] = e;
   }
   public void displayList()
   {
      for (int i=0;i<nbrStudents; i++)
         list[i].display();
   }
}
```



3.1. Usage and Advantage of Polymorphism

- Polymorphism
  - Using polymorphism in combination with dynamic linking
    - need to distinguish different cases depending on the class of objects.
    - possible to define new functionality by inheriting new data types from a common base class without having to modify the code that handles the base class interface
  - Faster development
  - The greater simplicity and better code organization
  - Easily expandable programs
  - Easier code maintenance

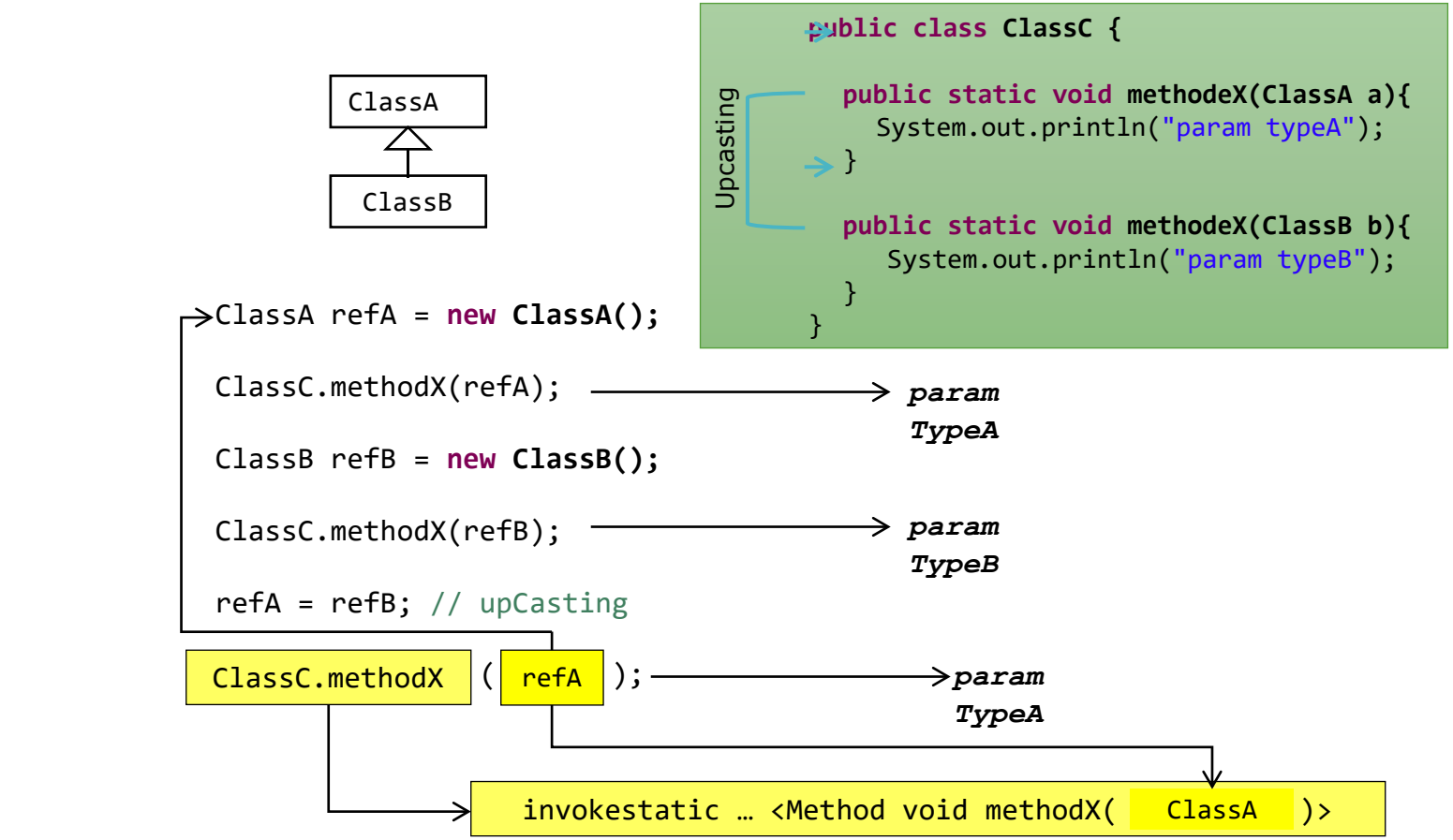
3.1. Usage and Advantage of Polymorphism

- Polymorphism
  - To conclude

« Once you know that all method binding in Java happens polymorphically via late binding, you can always write your code to talk to the base class, and know that all the derived-class cases will work correctly using the same code. Or put it another way, you send a message to an object and let the object figure out the right thing to do »

Bruce Eckel, Thinking in Java

3.2. Upcasting and polymorphism



The choice of the method to execute is done at compile time based on the declared class: Static Selection

3.3. About Method equals()

- Test the equality of two objects of the same class

```
public class Object {
   ...
   public boolean equals(Object o)
     return this == o
   }
   ...
}
```

```
public class Point {

   private double x;
   private double y;

   ...

}
```

```
public boolean equals(Point pt) {
  return this.x == pt.x && this.y == pt.y;
}
```

In general, it is best to avoid overloading methods specializing in the arguments

Overload : the method equals (Object o) inherited from Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)  --> true
Object o = p2;
p1.equals(o)   --> false
o.equals(p1)   --> false
```

The choice of the method to execute is done statically at compile depending on the type declared of object receiving the message and the type declared of (the) parameter (s)

3.3. About Method equals()

- Test the equality of two objects of the same class

```
public class Object {
   ...
   public boolean equals(Object o)
     return this == o
   }
   ...
}
```

```
public class Point {

   private double x;
   private double y;

   ...

}
```

```
public boolean equals(Object o) {
   if (! (o instanceof Point))
      return false;
   Point pt = (Point) o; // downcasting
   return this.x == pt.x && this.y == pt.y;
}
```

overrides the method equals
(Object o) inherited from Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)  --> true
Object o = p2;
p1.equals(o)   --> true
o.equals(p1)   --> true
```

4. Downcasting

```
ClassX obj = …
ClassA a = (ClassA) obj;
```

- Class ClassX and ClassA are 2 different classes
- How could this be done?
- Do you used to convert double to an int or int to a float?

4.1. Force to a kind

- The downcasting (or casting) allows you to "force to a kind" on compilation
  - It is a "promise" that we made at the time of compilation.
- For the cast to be valid, it must be effective and the execution type of obj is "compatible" with the type ClassA
  - Compatible: the same class or any subclass of ClassA

  (obj instanceof ClassA)
- If the promise is not held a runtime error occurs.
  - ClassCastException up and stop running

```
java.lang.ClassCastException: ClasseX
        at Test.main(Test.java:52)
```

4.2. Compatible class

- When you try casting to an incompatible class, it produce this at runtime:

```
java.lang.ClassCastException: ClasseX
        at Test.main(Test.java:52)
```

- The compatible classes are:
  - Classes which is subclass of casted class (upcasting)
  - Classes which is superclass of casted class (downcasting)
  - Type that are bigger types such as char, int, float and double

4.3. Example

- Simple types:

```
int a = 5;
double x = a;
float y = (float)x;
```

- Structure types:

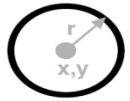
```
public class Main {
  public static void main(
          String[] args) {
    B b = new B();
    A a = (A)b;
  }
}
```

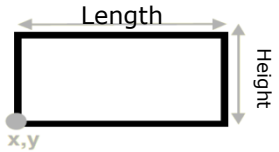
```
public class A {
  public void hello() {
    System.out.println("Hello");
  }

  public void display() {
    System.out.println("I am A");
  }
}
```

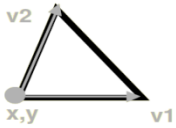
```
public class B extends A {
  public void display() {
    System.out.println("I am B");
  }
}
```

5.1. Abstract class example

- a classic geometric shapes
  - we want to define an application for working with geometric shapes (triangles, rectangles, circles ...).
  - each shape is defined by its position in the plane
  - each form can be moved (changing its position), can be calculated its perimeter, its surface







Attribute:
double x, y; // center of the circle
double r ; // Ray

Methods:
move (double dx, double dy)
double surface ()
double perimeter ()

Attribute:
double x, y; bottom left corner
double width, height;

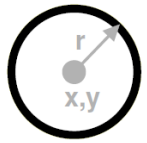
Methods:
move (double dx, double dy)
double surface ()
double perimeter ();

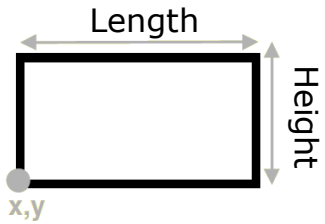
Attribute:
double x, y; // 1 of vertices
double x1, y1; // v1
double x2, y2; // v2

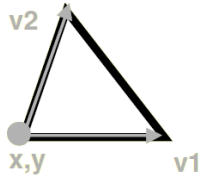
Methods:
move (double dx, double dy)
double surface ()
double perimeter ();

  - Refactor the code?

5.1. Abstract class example

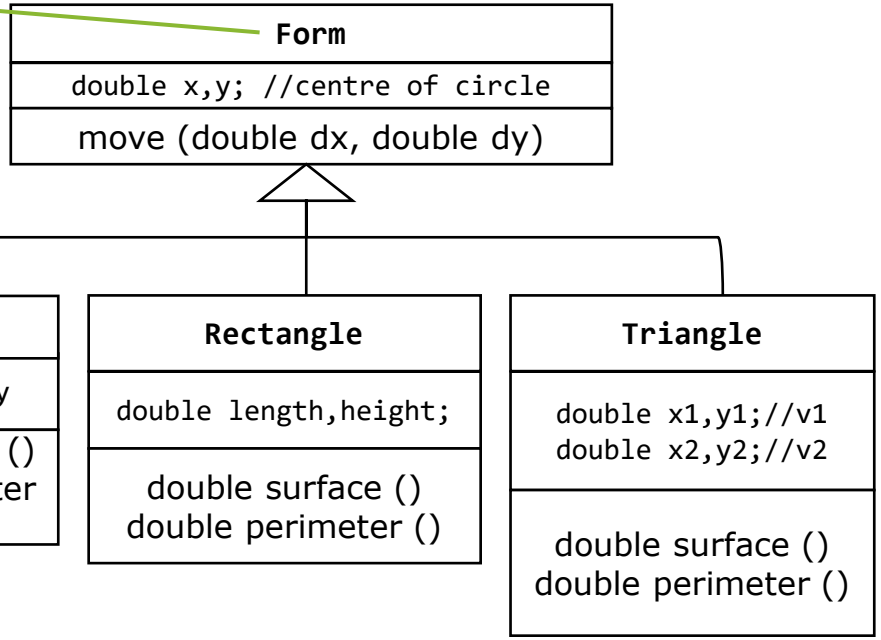







```
class Form {

  protected double x,y;

  public void move(double dx,
                   double dy) {
    x += dx; y += dy;
  }
}
```

```
class Circle extends Form {

  protected double r;
  public double surface(){
    return Math.PI * r * r;
  }
  protected double
  perimeter(){
    return 2 * Math.PI * r;
  }
}
```



5.1. Abstract class example

```
public class ListOfForms {
  public static final int NBR_MAX = 30;
  private Form[] tabForm = new Form[NBR_MAX];
  private int nbrForms = 0;

  public void add(Form f)
  {
    if (nbrForms < NBR_MAX)
        tabForm[nbRForms++] =f;
  }
  public void allMove(double dx,double dy)
  {
    for (int i=0; i < nbrForms; i++)
        tabFrom [i].move(dx,dy);
  }

  public double perimeterTotal()
  {
    double pt = 0.0;
    for (int i=0; i < nbrForms++; i++)
        pt += tabForm[i]. perimeter();
    return pt;
  }
```

It operates the decision **polymorphism** into account of new types of form does not modify the code

Call invalid because the perimeter method is not implemented at the Form class

Set a perimeter method in Form?

```
public double perimeter ()
{
  return 0.0; //or -1. ??
}
```

A clean and elegant solution: abstract classes

5.2. Requirement to use Abstract Classes

- define incomplete concepts that will be implemented in the subclasses
- refactor the code

Class abstract

Abstract methods

```
public abstract class Form {

  protected double x,y;

  public void translate(double dx,
        double dy) {
    x += dx; y += dy;
  }

  public abstract double length() ;

  public abstract double surface();
}
```

5.2. Requirement to use Abstract Classes

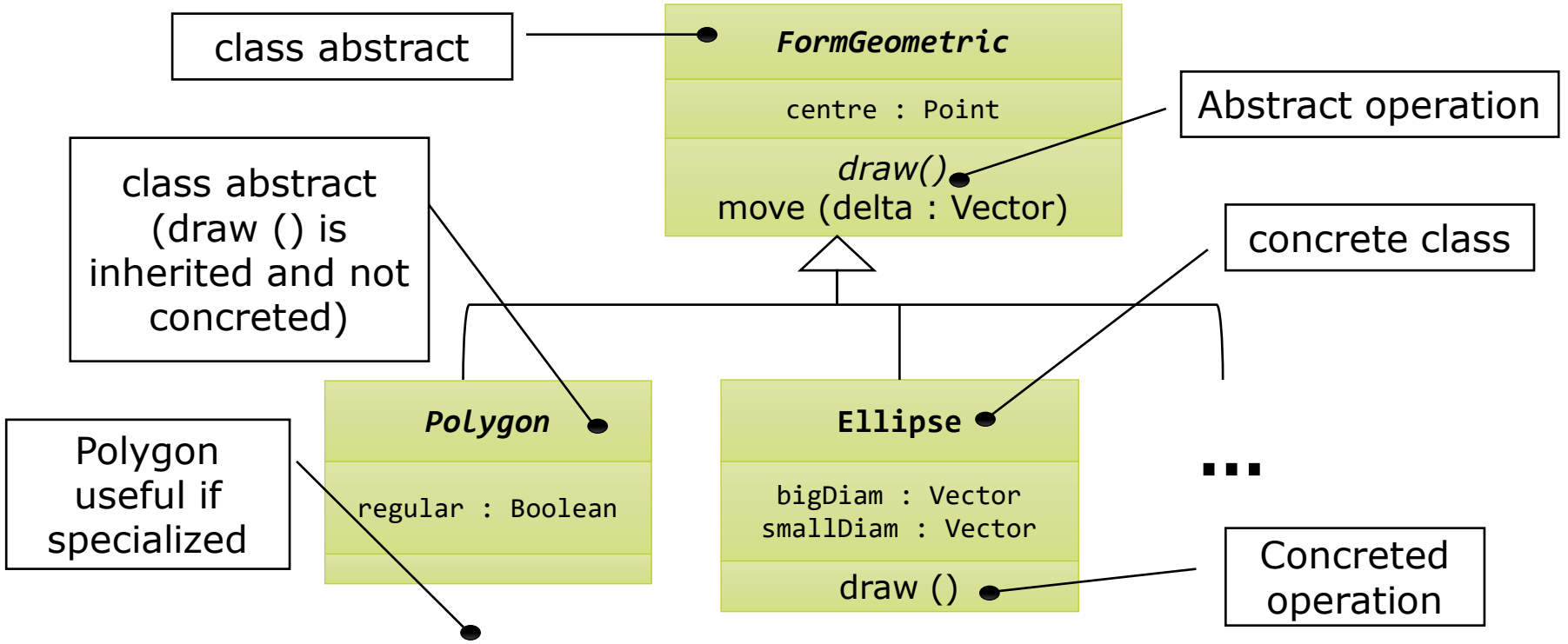
- **abstract class:** not instantiable class, that is to say it admits no direct instances.
  - Can not make new ClasseAbstraite (...);
- abstract operation: Operation not admitting implementation
  - at the class in which it is declared, we can not realize how to achieve it.
- A class for which at least one is declared abstract operation is an abstract class (the reverse is not true).
- The abstract operations are particularly useful for performing polymorphism.
  - using the name of an abstract class as the type for (the) reference (s) is always possible (and often desirable !!!)

5.2. Requirement to use Abstract Classes

- An abstract class is a description of objects to be inherited by more specialized classes.
- To be useful, an abstract class must admit concrete descendant classes.
- Any concrete class subclass of an abstract class must "fulfill" all abstract operations of the latter.
- An abstract class is used to group certain features common to its subclasses and defines a minimum common behavior.
- The optimal factorization properties common to several classes a generalization usually requires the use of abstract classes.

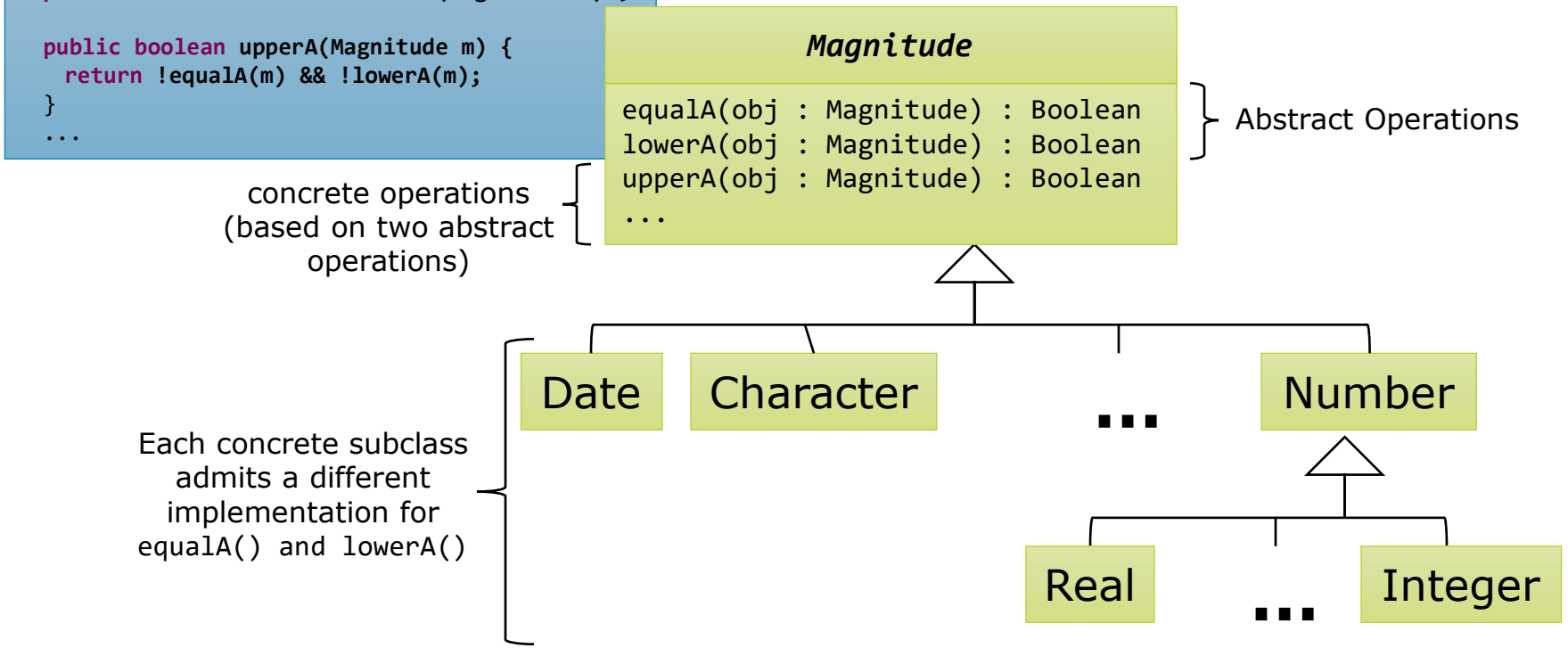
5.3. Better structure of above Example

- Better structure with classes and abstract operations



5.3. Better structure of above Example

```
public abstract class Magnitude {
  public abstract boolean equaA(Magnitude m) ;

  public abstract boolean lowerA(Magnitude m) ;

  public boolean upperA(Magnitude m) {
    return !equalA(m) && !lowerA(m);
  }
  ...
```



Test

Question	Possible Answers	Correct Answer
Filling in the gaps:	Upcasting is action of an object of a …… type can be treated as an object of any …… type.	
Filling in the gaps:	……Action that an object of a superclass type to be treated as an object of any …… type.	
Which is not dynamic-link?	a) Late binding b) Runtime binding c) Dynamic rating d) Dynamic binding	

Test

Question	Possible Answers	Correct Answer
We are in Class ClassA, to override method equals(), we create method:	a) boolean equal(ClasssA a) b) boolean equal(Object o) c) boolean equals(Object o) d) boolean equals(ClassA a)	
Abstract class is:	a) Class that has all abstract methods b) Class that declared with keyword abstract c) Class that has abstract method(s)	

Practice

Excercises
1. Practices class **`ListOfForms`** in this course.
2. Create a class Fish that represents current known fish types in Tonle Sap river.
3. Create class FishDictionary that user can find fish by name.

Summarize

- Upcasting allows an object of a subclass type to be treated as an object of any superclass type.
- Dynamic-link allows resolving messages of an object at runtime.
- Polymorphism allows an object to appear in many forms (types or classes)
- Downcasting allows an object of a superclass type to be treated as an object of any subclass type.
- Abstract class is class that may have abstract method(s) and can not be directly instantiated.

Reference

- "Object Oriented Programming" by Ph. Genoud – Université Joseph Fourier, 2006
- "Building Skills in Object-Oriented Design" by Steven F. Lott, 2009
- "Exercices en Java" 2nd edition by Groupe Eyrolles, 2006
- "Java Programming – Introductory" by Joyce Farrell, 1999
- "Java Examples in a Nutshell" 3rd Edition by David Flanagan, 2004
- "Programmer en Java" 5th edition by Groupe Eyrolles, 2008