Data Structures

Augustin Cosse.

Spring 2021

February 9, 2021

Package import

- As we saw, every stand-alone public class defined in java must be given a separate file. The file name is the name of the class with a .java extension.
- A class declared as `public class Window` will be defined in the file `Window.java`. That file may contain definitions of other standalone classes but none of them may be declared with public visibility
- To help with the organization of large repository, Java allows a group of related type definition (such as classes or enums) to be grouped into what is known as as a package
- For types to belong to a package named `packageName`, their source code must all be located in a directory named `packageName` and each file must begin with the line **package** packageName

Package import

- To refer to a type within a package name, we can use the dot notation with the type treated as an attribute of the package. For example, if the Window class is defined in the package architecture, we can declare a variable with **architecture.Window** as its type.
- Packages can be organized in subpackages (which then have to be grouped within a subdirectory of the package directory)
- The dot notation can also be used with subpackages. As an example, the **java.util.zip** subpackage within the **Java.util** package.

Package import

- The use of packages has several advantages:
  - Package can reduce name conflicts. If all the files are in a single folder, there can be only one class with the name **Window**. But with packages, we can imagine having a class saved as architecture.Window and another saved as gui.Window
  - The distribution of the code is easier (the code will be more easily understood and used by other programmers)
  - Recall that clases within the same package have access to any of each others members having anything but private (i.e. public, protected or default) visibility

Package import

- We can always refer to a type stored in a package through its "fully qualified name" (e.g. the Scanner class is defined in the java.util package and we can refer to it as java.util.Scanner)

- We can thus always declare and construct a new instance of that class in a project using the statement

```
java.util.Scanner input =
                    new java.util.Scanner(System.in);
```

- To be more efficient, java allows the use of the keyword **import** to include external classes or entire packages in the current file. To import a class from a specific package, we use the line below

```
import packageName.className;
```

Package import

- As an example, to import and to use the class Scanner, we can then use

```
import java.util.Scanner;
Scanner input = new Scanner(System.in)
```

- In order to import all the definitions from a given package, one can also use the line

```
import packageName.*;
```

such as in

```
import architecture.*;
import gui.*;
```

Software development

- Traditional software development relies on the following main steps
  - Design
  - Coding
  - Testing and Debugging
- For object oriented programming, the design step is perhaps the most important phase. It is in the design step that we decide how to divide the workings of our program into classes, and decide how those classes will interact, what data they will store and how they will interact.

Object oriented design

- The main actors in the object oriented paradigm are called objects
- Each object is an instance of a class.
- The class specifies the instance variables that the objects will contain as well as the methods that the object will be able to execute.
- Software development should achieve robustness, adaptability and reusability

Object oriented design

- In general, we will want our programs to be robust (i.e. capable of handling unexpected inputs that are not explicitely defined for its application)
- For example, if a program is expecting positive integers and it is given a negative integer instead, the program should be able to gracefully recover from this error.
- In life critical applications, where a software error can lead to injury or loss of lives, a software that is not robust could be deadly.
- As an example, in the 1980's, a radiation therapy machine severely overdosed 6 patients (becasue of software errors), some of whom died from complications resulting from the radiation overdose.

Object oriented design

- Modern software development such as web browsers involve large programs that are used for many years.
- These programs therefore need to be able to evolve over time in response to changing conditions in the environment
- Another important goal in software development is for the programs to achieve adaptability
- An particular aspect of this is portability (i.e. the ability of software to run with minimal changes on different harware and operating system platforms)

Object oriented design

- Finally, a last important aspect of efficient software development is reusability.
- The same code should be usable as a component of different systems in various applications
- Developing high quality softwares can be expensive and the cost of software development can thus be reduced significantly if the sotfware is designed in a way that makes it reusable in future applications.

Object oriented design

- Object oriented design relies on the following principles
  - Abstraction
  - Encapsulation
  - Modularity

Abstraction (I)

- The notion of abstraction consists in splitting a complicated system into its most fundamental components
- Describing the parts of a system involves naming them and explaining their functionalities
- Applying the idea of abstraction to the design of data structures gives rise to abstract data types.
- An Abstract Data Type specifies what each operation does but not how to do it
- An ADT can be expressed by an interface which is simply a list of method declarations where each method has an empty body

Abstraction (II)

- An abstract data type is realized by a concrete data structure which in Java is modelled by a class.
- Unlike interface, classes specify how the operations are performed in the body of each method
- A java class is said to implement an interface if its methods include all the methods declared in the interface
- A class can have more methods than the interface

Encapsulation

- The idea of encapsulation is that the different components of a program should not reveal the internal details of their respective implementation
- Encapsulation gives a programmer the freedom to implement the details of a component without concerns that other programmers might be writing code that intricately depends on those internal decisions
- Encapsulation yields robustness and adaptability (i.e. implementation details of parts of a program can be changed without affecting other parts, making it easier to fix bugs)

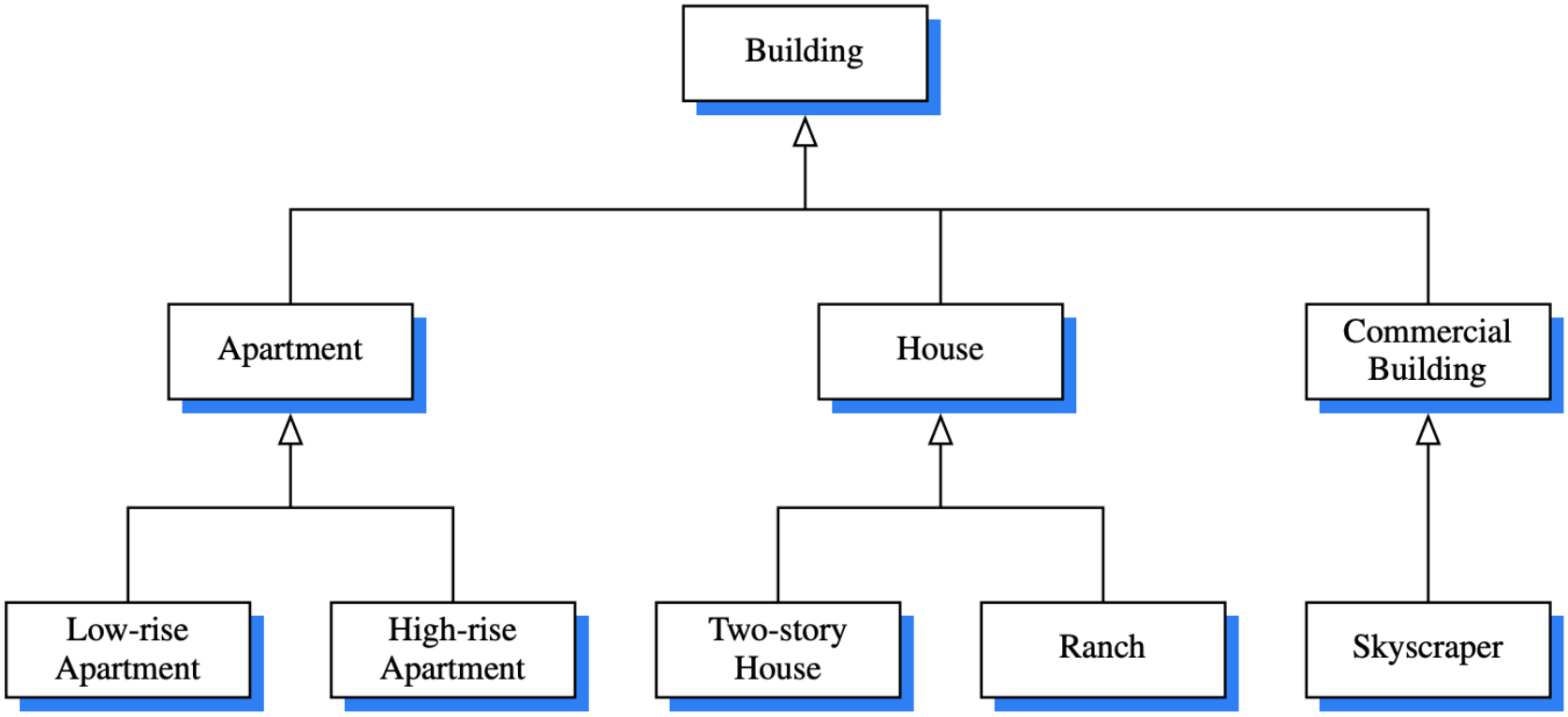
Modularity

- Modularity refers to the idea that different components of a software should be divided into separate functional units.
- Modularity improves robustness beacuse it is easier to test and debug separate components before they are integrated into a larger software system.

Inheritance

- A natural way to organize the various components of a software package is in a hierarchical fashion with abstract definitions grouped in a level-by-level manner
- Ex: The set of houses is a subset of the set of buildings but a superset of the set of ranches
- We will usually refer to such as correspondance through the "is a" relationship. A house is a building and a ranch is a house.

Inheritance



Inheritance

- In object oriented programming, the mechanism for a modular and hierarchical implementation is implemented through inheritance
- A new class can be defined based upon an existing class as the starting point
- In object oriented programming, the existing class is typically described as the base class, the parent class or the super-class, while the newly defined class is known as the subclass or chilc class
- We say that the subclass extends the superclass.

Inheritance

- When inheritance is used, the subclass automatically inherits, as its starting point, all the methods from the superclass (other than constructors)
- The subclass can differentiate itself from its superclass in two ways. It may augment the superclass by adding new fields and new methods and it may also specialize existing behaviors by providing a new implementation that overrides an existing method

Inheritance

- As an illustration of inheritance, consider the following class

```
public class CreditCard{

private String customer;
private String bank;
private String account;
private int limit;
protected double balance;

// constructor
public CreditCard(...){}
// A method
public boolean charge(double price){ ... }
// a second method
public void makePayment(double amount){ ... }
}
```

Inheritance

- A child of that class could be given by the following Predatory class

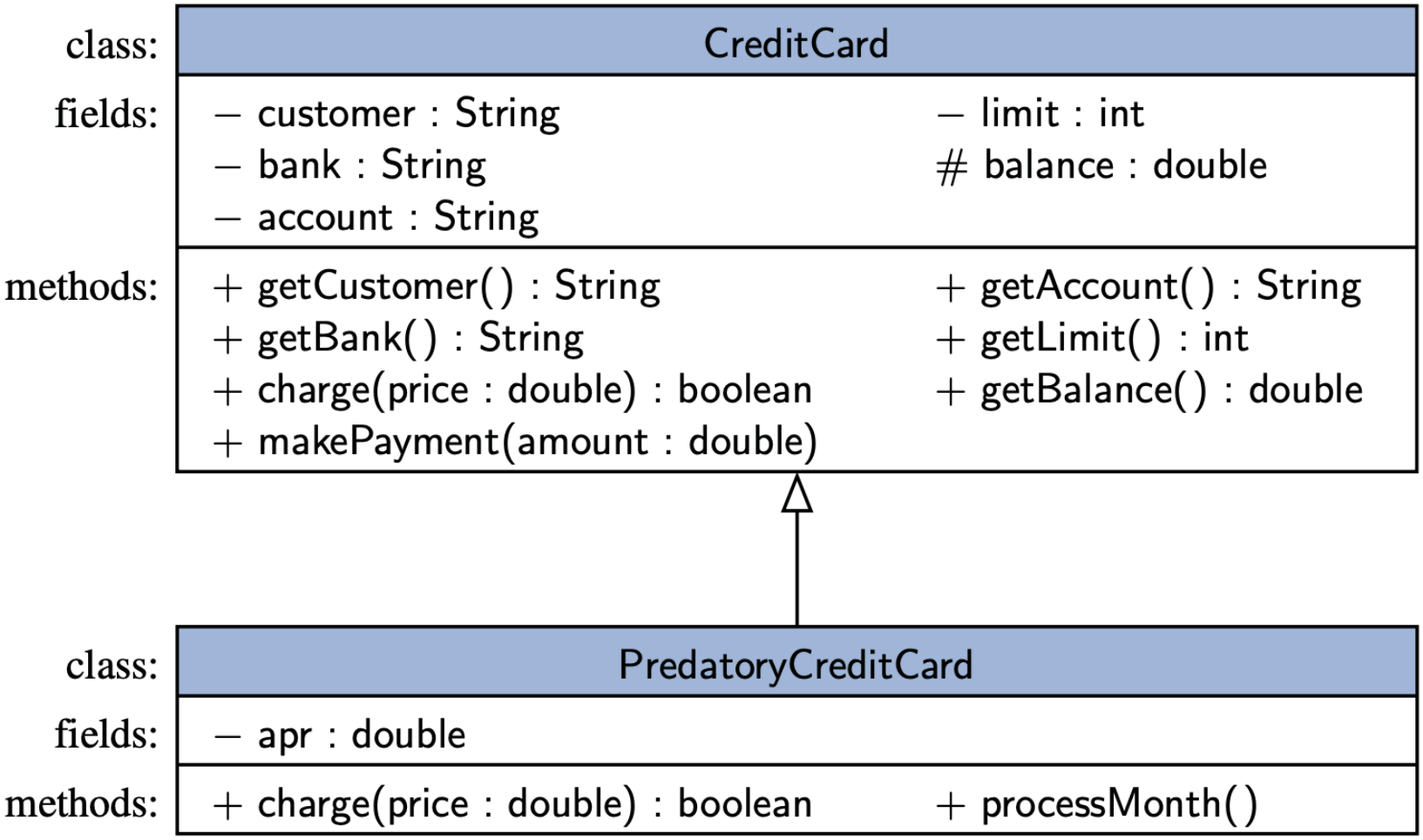
```
public class PredatoryCreditCard extends CreditCard{

private double apr;

// constructor for the class
public PredatoryCreditCard(String cust)
// a new method
public void processMonth(){...}
// Overriding the charge method defined
// in the parent class
public void boolean charge(double price){


}
```

Inheritance



Inheritance

- The predatory class augments the original CreditCard class, adding a new instance variable named **apr** and adding a new method named **processMonth**. The new class also specializes its superclass by overriding the original charge method in order to provide a new implementation.

- You see that to indicate that the new class inherits from the existing CreditCard class, we use the keyword **extends**. followed by the name of the parent class. In Java each class extends exactly one other class (We say that Java only allows single inheritance among classes)

```
public class PredatoryCreditCard extends CreditCard{

...

}
```

Inheritance

- Note that if even if a class makes no explicit use of the extends keyword, it automatically inherits from a class **java.lang.Object** which serves as the universal superclass in Java

- On top of newly defined **apr** variables, the child class PredatoryCreditCard will also inherit all the variables (customer, bank,a ccount, limit, and balance) from its parent class

```
public class PredatoryCreditCard extends CreditCard{

private double apr;
...
}
```

Inheritance

- Constructors are never inherited in Java. When the child class is created, all of its fields must be properly initialized (including the inherited fields). For this reason, the first operation performed within the body of a constructor of a child class is usually to invoke a constructor from the superclass.

- In Java the constructor from the superclass can be invoked by using the keyword **super** with the appropriate parametrization.

```
public class PredatoryCreditCard extends CreditCard{
  public PredatoryCreditCard(String cust, String bk,
                             String acnt,...){
  // invoking the constructor from the superclass
  super(cust, bk, acnt,..);
  apr = rate; }
}
```

Inheritance

- The use of the **super** is very similar to the use of the **this** used to invoke a different constructor from a given class.
- If a constructor of a child class does not make an explicit call to **super** or **this**, an implicit call will be made by Java through the command **super()** (call to the zero parameter constructor).
- On top of the call to **super()**, you see that we also need to initialize the new variable **apr**

```
public class PredatoryCreditCard extends CreditCard{
  public PredatoryCreditCard(String cust, String bk,
                             String acnt,...){
  // invoking the constructor from the superclass
  super(cust, bk, acnt,..);
  apr = rate; }
}
```

Inheritance

- On top of the methods that it inherits form the parent's class, you can see that the child class PredatoryCreditCard also defines a new method **processMonth()**

- You also see that the new method accesses the variable balance. This is permitted because that attribute was declared with **protected** visibility in the original CreditCard class.

```
public class PredatoryCreditCard extends CreditCard{
  public PredatoryCreditCard(String cust, String bk,
                             String acnt,...){}

  public void processMonth(){
    if (balance>0){
      ...

}}}
```

Inheritance

- Finally, through the lines below, we can also override the method **charge** which is inherited from the parent class.

- You also see that within our new implementation of the method we can still use some of the attributes from the parent class through the keyword **super** (here **super**.charge())

```
public class PredatoryCreditCard extends CreditCard{
  public PredatoryCreditCard(String cust, String bk,
                             String acnt,...){}

  public void processMonth(){}
  public boolean charge(double price){
      boolean isSuccess = super.charge(price);

    ...

}}
```

Polymorphism

- The word polymorphism literaly means "many forms". In the framework of object oriented programming it refers to the ability of a reference variable to take many forms.
- This idea is known as the Liskov Substitution principle (it is a manifestation of the "is a" relationship)
- We say that the variable **card** is polymorphic. Note that if a variable is declared with a specific type, it will only be able to use the methods that are defined at the level of that type.

```
CreditCard card;
CreditCard card = new PredatoryCreditCard(...);
```

Polymorphism

- In the example below, since the object is a **PredatoryCreditCard** instance, it will execute the **PredatoryCreditCard.charge** method even if the referenced object has been declared with the type **CreditCard**

```
CreditCard card;
CreditCard card = new PredatoryCreditCard(...);
```

Polymorphism

- Now one could wonder what happens if we consider the call to **card.charge()** (given that both **CreditCard** and **PredatoryCreditCard** have their own implementation of this method.)
- In such a situation, Java relies on a process called dynamic dispatch.
- The idea of dynamic dispatch is that java decides at runtime to call the version of the method that is most specific to the actual type (not the declared type) of the referenced object

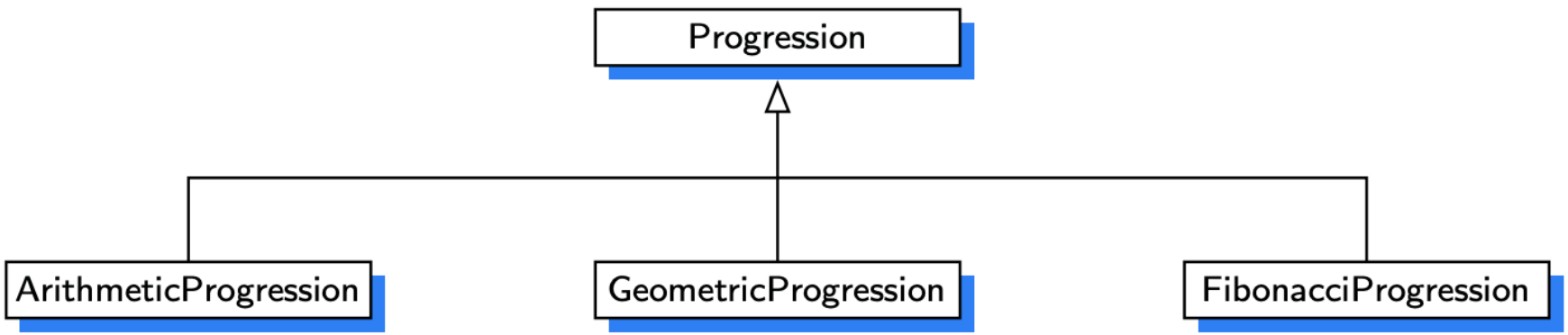
```
CreditCard card;
CreditCard card = new PredatoryCreditCard(...);
```

Inheritance

- Java also provides an **instance of** operator that tests at runtime whether an instance satisfies as a particular type.
- As an example the line 'card **instance of** PredatoryCreditCard' produces **true** if the object currently referenced by the variable card belongs to the PredatoryCreditCard class or any further subclass of that class.

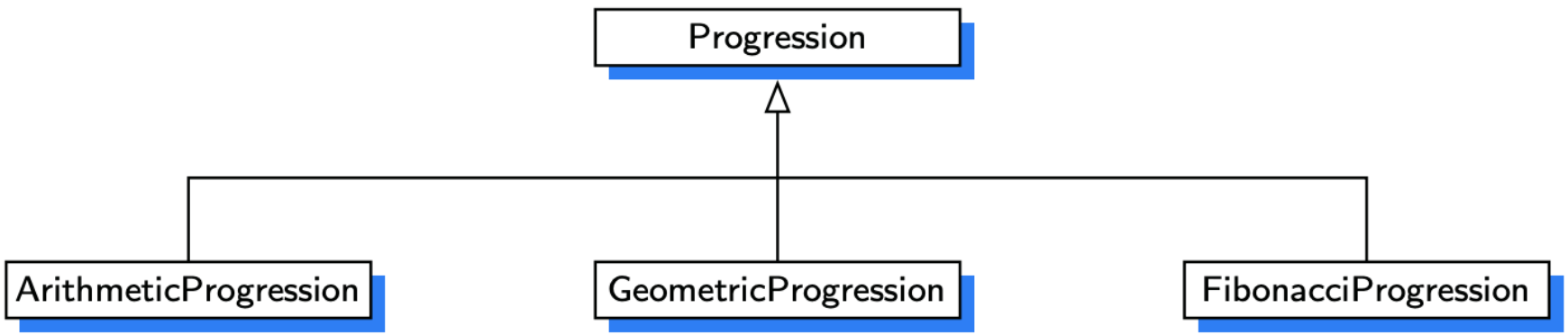
Inheritance hierarchies

- Although a subclass cannot inherit from multiple superclasses in Java, a superclass may have many subclasses. It is in fact quite common in Java to develop complex inheritance hierarchies to maximize reusability of the code
- As an illustration of this, we consider the following inheritance diagram for Progression:



Inheritance hierarchies

- A general progression is a sequence of numbers, in which each current number depends on one or more of the previous numbers.
- An arithmetic progression determines the next number by adding a fixed constant to the previous value.
- A geometric progression determines the next number by multiplying the previous value by a fixed constant.



```
public class Progression{

  // instance variable
  protected long current;

  /** Constructs a progression starting at zero */
  public Progression() {this(0); }

  /** Constructs a progression with given start value*/
  public Progression(long start){ current = start; }

  /** Returns the next value of the progression */
  public long nextValue() {
   long answer = current;
  advance(); // this protected call is responsible for
                      // advancing the current value
  return answer();

...
}}
```

```
public class Progression{
...
  /** Advances the current value to the next value
of the progression */
  protected void advance(){
    current ++;
  }
  /** Prints the next n values of the
progression, separated by spaces */

  public void printProgression(int n){
    System.out.print(nextValue());
        // print first value w/o space
    for(int j=1; j < n; j++)
       System.out.print(" " + nextValue());
        // print leading space before the others
      System.out.println();
}}
```

Inheritance hierarchies

- The class Progression has a single field named **current**. It also defines two constructors, one accepting a specified starting value for the progression and the other using the default 0 value.
- On top of those, the class also has the remaining attributes:
  - **nextValue()** : A public method that returns the next value of the progression, also implitely advancing the progression each time.
  - **advance()**: A protected method responsible for advancing the value of current in the progression
  - **printProgression(n)**: A public utility that advances the progression n times while displaying each value

Inheritance

- The method **advance()** in the Progression class is the one that we will want the children classes to override.
- We consider three subclasses of the Progression class: ArithmeticProgression, GeometricProgression and FibonacciProgression.

Inheritance

- As a first example of a child class, we consider the class ArithmeticProgression
- An arithmetic progression adds a fixed constant to the current term of the progression to produce the next
- ArithmeticProgression relies on the class **Progression** as its parent class
- It includes three constructors (one of them calling the superclass constructor through **this**) and override the protected advance method

```
public class ArithmeticProgression extends Progression
{   protected long increment;
    /** Constructs progression 0,1,2,...*/
    // start at 0 with increment of 1
  public ArithmeticProgression() {this(1,0); }
    /** Constructs progression 0, stepsize,
         2*stepsize, ...*/
  // starts at 0
public ArithmeticProgression(long stepsize) {
this(stepsize,0);}
  /** Constructs arithmetic progression with
       arbitrary start and increment */
public ArithmeticProgression(long stepsize, long start)
{   super(start);
    increment =  stepsize; }
  /** Adds the arithmetic increment to the current
       value */
  protected void advance(){
     current +=increment;
}}
```

Inheritance

- As a second example of a child class, we consider the class GeometricProgression
- The class introduces one new field: the base of the geometric progression
- It also provides three constructors, just as the ArithmeticProgression class, for convenience.
- Finally, it overrides the protected **advance** method

```
public class GeometricProgression extends Progression{
   protected long base;
   /** Constructs progression 1,2,4,...*/
   // start at 1 with base of 2
  public GeometricProgression() {this(2,1); }
   /** Constructs progression 1, b, b^2, ...*/
  // starts at 1
  public GeometricProgression(long b) {this(b,1 );}
  /** Constructs geometric progression with
        arbitrary base and start */
  public GeometricProgression(long b, long start){
    super(start);
    base = b;
}
  /** Multiplies the current value by the
       geometric base */
  protected void advance(){
     current*=base;
}}
```

Inheritance

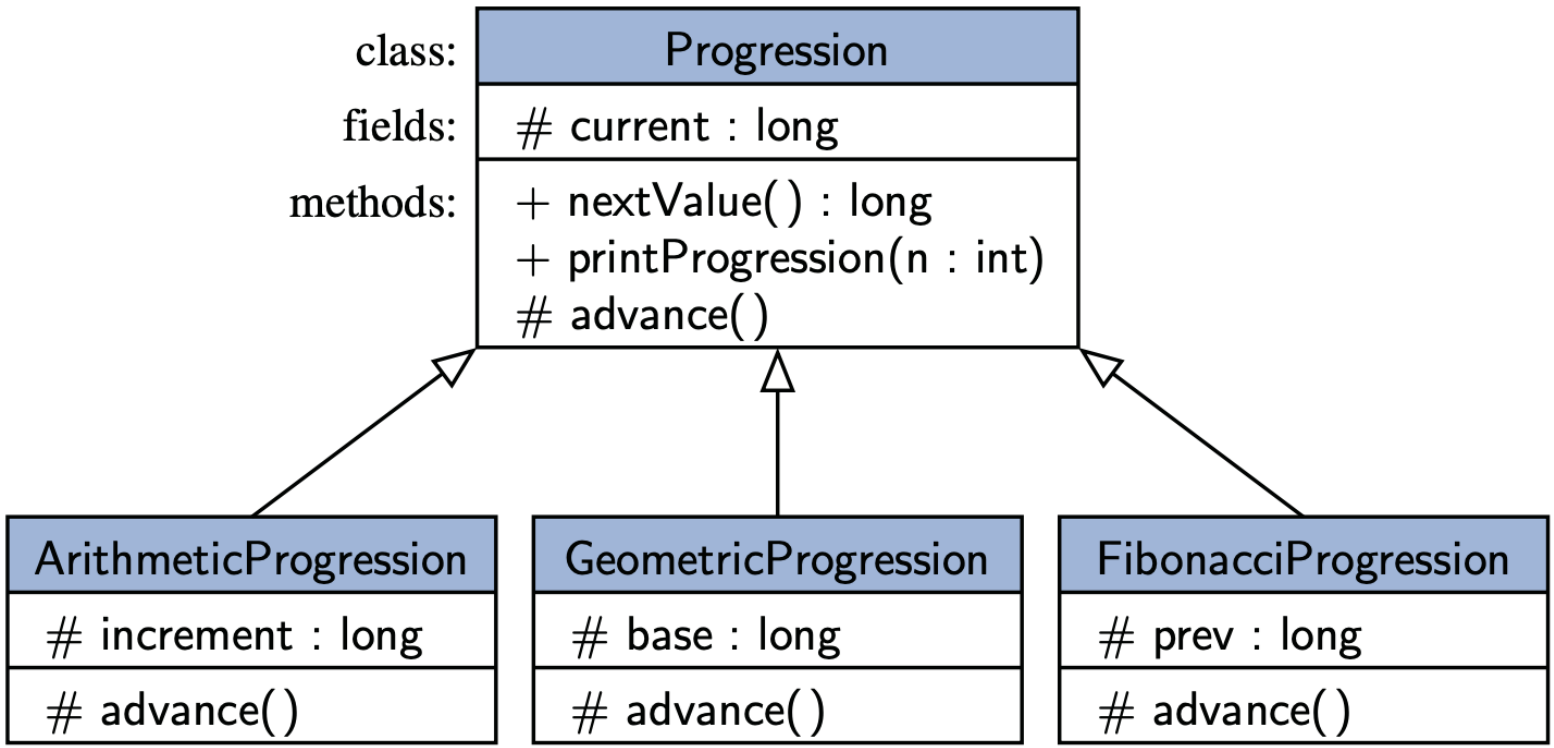
- Finally, as a last example of a child class, we consider the class **FibonacciProgression**
- Each value of a Fibonacci series is the sum of the two most recent values. To begin the first two values are conventionally 0 and 1
- The FibonacciProgression class is stored as an arithmetic Progression with a varying increment
- The Fibonacci class introduces a new value **prev** which stores the value F_{n-2} such that $F_n = F_{n-1} + F_{n-2}$. The value of **prev** is initialized as $F_0 = F_2 - F_1$

```
public class FibonacciProgression extends Progression{
  protected long prev;
   /** Constructs Fibonacci .*/
   // starting 0,1,1,2,3
  public FibonacciProgression() {this(0,1); }
   /** Constructs generalized Fibonacci */
  // with given first and second value
  public FibonacciProgression(long first, long second) {
  super(first);
  prev = second - first; }
// fictitious value preceding first
  /**Replaces (prev, current) with
 (current, current+prev) */
  protected void advance(){

   long temp = prev;
   prev = current;
  current + = temp;
}}
```

Inheritance

- The three classes can then be represented through the following inheritance diagram



Interfaces and abstract classes

- In order for two objects to interact, they must know the methods that each supports.
- To enforce this knowledge, Object Oriented Programming requires the classes to specify the Application Programming Interface (API).
- In the framework of this course, we will define our interface as a type definition and a collection of methods for this type with the arguments for each method being of specified types.
- The specification will then be enforced by the compiler which will require that the types of the parameters that are passed to the method rigidly conform with the type specified in the interface. (Such a requirement is known as strong typing)

Interfaces and abstract classes

- Having to define interfaces might seem to place a burden on the programmer. Yet this burden is often offset by the fact that it will make it possible to catch programming error that would otherwise go unnoticed.
- Interfaces do not have constructors
- When a class implements an interface, it must implement all the methods declared in the interface. (Note that additional methods could be added)
- The interface should be stored in a separate .java file.

```java
/** An example of an interface for objects
 that can be sold  */
public interface Sellable{
   /** Returns a description of the object */
  public String description();

  /** Returns the list price in cents */
 public int listPrice();


  /** Returns the lowest price in cents we
       will accept */
 public int lowestPrice();
}
```

```java
/** An implementation of the interface Sellable  */
public class Photograph implements Sellable {
  private String descript; // photo descript.
  private int price; // price
  private boolean color; // true if color photo

  public Photograph(String desc, int p, boolean c){
// constructor
 descript = desc;
price = p;
color = c; }

  public String description() {return descript; }
  public int listPrice() {return price; }
  public int lowestPrice() {return price/2; }
  public boolean isColor() {return color; }
}
```

- Note that a class can implement multiple interfaces as illustrated by the class **BoxedItem** below

```
public interface Transportable {
...
}
```

```
public class BoxedItem implements Sellable,
                                  Transportable {
...
}
```

Interfaces and abstract classes

- The ability of extending from more than one type is known as multiple inheritance
- In Java, although multiple inheritance is not allowed for classes, it is allowed for interfaces
- The intuition behind this is that interfaces do not define fields or method bodies. If Java was allowing multiple inheritance for classes there could be a confusion if a class tries to extend from two classes with the same variables names or methods with the same signatures.

Interfaces and abstract classes

- One interest of using multiple inheritance for interface is to mimic the **mixin** technique from C++ or Smalltalk. Some object oriented all multiple inheritance from concrete classes as a way to provide particular functionalities to a given class
- This idea can be approximated with interfaces and the methods from a mix of interfaces can be used to define a new interface that combines their functionalities.

Abstract classes

- In Java an abstract class serves a role in between a traditional class and an interface
- Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies. Such methods are known as abstract methods
- However, unlike an interface, an abstract class may also define or more fields and any number of methods with implementations. (so-called concrete methods.)
- An abstract class may also extend another class and be extended by further subclasses.

Abstract classes

- As is the case with interfaces, an abstract class may not be instantiated. I.e no object can be created from the class.
- A subclass of an abstract class must therefore provide implementations for the abstract methods, or else remain abstract.
- A non abstract class will be called concrete class.
- Abstract classes are classes and hence limited to single inheritance. Abstract classes thus have at most one super class (concrete or abstract).

Abstract classes

- Abstract classes are particularly interesting as the support reusability of code
- In particular, commonality between concrete classes can be placed at the level of an abstract class that can serve as a super class.
- By using an abstract, we can then force a collection of classes to share a mixing of functionalities.
- The main point of the **Progression** class which we discussed before was to provide common functionalities for the three classes **ArithmeticProgression**, **GeometricProgression** and **FibonacciProgression** so that we could have instead, decided to define this class as an interface.

Abstract classes

```
public abstract class AbstractProgression {

   protected long current;
   public AbstractProgression() {this(0); }
   public AbstractProgression(long start)
 {current = start; }

  public long NextValue(){
     long answer = current;
     advance();
     return answer;}
  public void printProgression(int n){
      System.out.print(nextValue());
      for(int j=1; j<n; j++)
        System.out.print(" " + nextValue());
      System.out.println();}
protected abstract void advance(); // abstract method
}
```

Abstract classes

- Note the use of the abstract modifier on the class definition
- Note how the method **advance** is defined as abstract
- You might also notice that the class provide constructors, despite the fact that (since it is abstract) it cannot be instantiated. The constructors can however be invoked within the subclasses constructors using the **super** keyword

Abstract classes

- You can also see that the abstract method **advance** does not have any body. Moreover it is being called inside the body of the **nextValue()** function
- For abstract classes, it is perfectly legit to call an abstract method from within the body of another concrete method.
- This is an example of an object oriented design pattern known as template method pattern in which an abstract base class provides a concrete behavior that relies upon calls to other abstract behaviors.
- Once a class provides definitions for the missing abstract behaviors, the inherited concrete behavior will be well defined.

Abstract classes

- Finally it might seem as though abstract methods are equivalent to overriding empty methods. In fact the difference lies in the fact that, since the implementation of the abstract method is defered, you cannot call this method before it has been implemented, reducing, once again the risk of making mistakes.

Exceptions

- Exceptions are unexpected events that occur during the execution of a program
- Exceptions might result from unavailable resource, unexpected input from a user or simply a logical error from the programmer
- In Java, exceptions are objects that can be thrown by code that encounters an unexpected situation.
- The general methodology for handling exceptions is the **try catch** construct

Exceptions

- The syntax for a **try catch** is the following

```
try{
// guarded body

}catch(exceptionType1 variable1) {
// remedeBody1
}  catch(exceptionType2 variable2) {
// remedeBody2
}
```

Exceptions

- When facing a try catch, the Java Runtime environment starts by processing the guarded body of the try
- If no exception are generated during the execution, the flow of control continues wih the first line beyond the entire try catch statement
- If an exception is generated during the execution of the try body, the execution jumps to the catch whose exception type most closely matches the exception object itself
- The variable for this catch statement references the exception object itself, which can be used in the block of the matching catch statement
- Once execution of that catch block completes, control flow continues with the first statement beyond the entire try catch.

Exceptions

- Consider the following example

```
public static void main(String[] args){
  int n = DEFULAT;
  try {
   n = Integer.parseInt(args[0]);
   if(n<=0){
  System. out.println("n must be positive. Using default
  n = DEFAULT;}
 catch(ArrayIndexOutOfBoundsException e){
System.out.println("No argument specified for n. Using d
} catch=(NumberFormatException e){
  System.out.println("Invalid integer number. Using defa

}



}
```