# ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND GALWAY

*Lecture 1a*

Lecturer: Dr Patrick Mannion

## REQUIREMENTS

- Basic Java knowledge from CT874 (semester 1)
- We will revise basic Java concepts this week
- A revision session can also be scheduled where I answer student-submitted questions about basic Java concepts (if there is enough demand)

- You need access to a PC/laptop where you can install:
  - Java Development Kit >=8 (for the first 6 weeks of the module)
  - A Java IDE such as Eclipse (you should already have an IDE set up from CT874)
  - Jakarta EE (more details will be provided in the second half of the module)
  - A Jakarta EE IDE (more details will be provided in the second half of the module)

# IF YOU NEED HELP WITH LECTURE MATERIAL, ASSIGNMENTS, ETC.

- Ask questions about lecture content during lectures
- If you need help with the assignments, please approach one of the lab tutors during the lab sessions:

  Conor F Hayes

  Yaser Babukhan
- Use the **Discussion Forum** on Blackboard to ask questions outside scheduled lecture and lab times

- If you need to discuss matters of a personal nature (e.g. if you are unable to complete an assignment due to illness) please send an email to me at patrick.mannion@nuigalway.ie

## SCHEDULE

- 2 x 2h lectures per week, worth 10 credits (so this module is twice as large as most other modules in terms of content, workload, etc.)
- Lecture "A" on Tuesdays 4pm-6pm, Online
  Lecture "B" on Thursdays 2pm-4pm, Online
- Lab classes:
  - Mondays, 4pm-6pm, Online
  - Thursdays, 4pm-6pm, Online
- <u>Labs start in week 2 (15th Feb onwards) and finish in week 12</u>

## ATTENDANCE

- Attendance is not compulsory for lectures, but you are encouraged to attend as many as possible
- Lecture recordings will be made available for download

- Attendance at labs is only required if you have a question regarding assignments, or software configuration and other technical issues

## LECTURE NOTES

- You can download the lecture notes as .pdf files from Blackboard from the "Learning Materials" section. Normally, they will be posted shortly before the respective lecture
- Sometimes I might update the notes some time after the lecture. I will add a note on Blackboard if the slides for a lecture have been updated.

## ASSESSMENT BREAKDOWN

- There will be five assignments
- Each assignment will be worth 8% (5 x 8 = 40%)
- There will be two MCQs worth 5% each
- There will be an open book exam after teaching is over, worth 50%

## ASSESSMENT

- I will email an announcement to you via Blackboard each time a new assignment is posted.
- You should submit solutions by the submission deadline via Blackboard.
- Your submissions will be graded by the lab tutors, normally within ca. 1-2 weeks.
- Assignment submissions via email will not be accepted
- An indicative assessment schedule is available on Blackboard in the "Assessments" section

## ASSIGNMENTS

- Please note that assignment submission deadlines are strict.
- Late submission <u>only with a medical or counsellor's certificate</u> (otherwise 0 marks!)

## ASSIGNMENTS

- Assignments need to be completed individually (no group work). The graders (that is, the lab tutors) are advised to check all submissions for plagiarism.
- Plagiarism => 0 marks for both involved parties (copier & copied) on entire assignment
- You can read more about plagiarism here: https://www.nuigalway.ie/plagiarism/

## EXAM

- There will be a written open-book exam (2 hrs.) worth 50% at the end of the semester.
- All lecture material is examinable, unless explicitly announced otherwise.
- Past exam papers from the 2018-2019 and 2019-2020 academic years are the most relevant for revision.
- 2017-2018 and older exams had a different structure, so they are not as relevant.
- Search for module codes CT545, CT875 at https://www.mis.nuigalway.ie/regexam/paper_index_search_form.asp
- Final information about the exam structure will be provided in the last teaching week (revision lecture)

## LITERATURE

- Literature for weeks 1-6 (not required but useful):
  - For Java in general: Cay S. Horstmann: Core Java, Volume II--Advanced Features (10th Edition or higher)
  - Specifically the new Java 8 features:
    Raoul-Gabriel Urma et al: Java 8 in Action: Lambdas, Streams, and functional-style programming (1st edition or higher)

## LITERATURE

- Java 8 API documentation:
  http://docs.oracle.com/javase/8/docs/api/

- Note: the current latest Java Enterprise Edition (Jakarta EE 9) is built on Java SE 8, we will focus on Java 8 concepts. Enterprise software developers require long support times (Java SE 8 is supported until 2030!). Newer Java SE editions usually have much shorter support times.

- The lecture of course cannot cover every Java class and method. Therefore, when completing the lab assignments, it will be necessary from time to time to look up things in the API documentation.

## REQUIRED SOFTWARE

- If you have difficulties installing or configuring Java 8 or Eclipse on your laptop, please approach one of the lab tutors during a lab session.

- Your assignment solutions in weeks 1-6 need to work with only Java SE >=8 (that is, without third-party libraries or other software), unless the respective assignment explicitly states an exception.

## LECTURE TOPICS

- In the first half of this semester, you will learn new Java concepts
- We will also cover concepts which you know already in more depth (e.g., I/O streams), and there will be some revision of Semester 1 content
- Starting in week 7, we will mainly focus on the basics of Enterprise Java (Jakarta EE, formerly known s Java EE)

## LECTURE TOPICS

- Planned topics (tentative - changes possible):
  - A closer look at polymorphism and exception handling, vs. Java 8 Optional
  - More about generics, functors
  - More about the Java Collections Framework (JCF)
  - Multithreading and concurrency control
  - Networking and more about I/O with Java
  - Java 8 Lambda expressions
  - Java 8 Streams
  - Enterprise Java features, e.g., JAX-RS API for creating RESTful web services
  - Servlets using Tomcat

## LECTURE TOPICS

- It is assumed that you already know elementary Java (e.g., objects, classes, basics of inheritance, interfaces, constructors, methods, variables, loops, basic modifiers like private, public, static, file I/O, lists and arrays, ...).
- However, if you have any questions about first-semester Java, the lab tutors and I will be happy to answer them.
- There will also be some amount of revision of Semester 1 content in this semester's lectures - however, we will generally go into more detail compared to Semester 1 (e.g., wrt. generics).

## THIS WEEK

- Some revision and a closer look at Java class inheritance, overriding and overloading, polymorphism, abstract classes/interfaces, exception handling,
- Also: Java 8's "Optional" values

## A NOTE ON CODING STYLE

- You are expected to follow common Java coding conventions in all of your assignment submissions
- Some % of marks in each assignment will be awarded for following good development practices, and complying with Java coding conventions
- Java code style conventions were first set out in a 1997 technical report from Sun Microsystems (this company was acquired by Oracle in 2010). This report is quite long so there is no need to read it in detail. Link: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
- A more concise version that includes most of the main points is available from the Apache Software Foundation ACE project. I recommend that you read all of this in detail. **You should follow the Apache guidelines when preparing your assignment submissions**. Link: https://ace.apache.org/docs/coding-standards.html
- Coding conventions do not affect the results of executing your code. However, it is important to follow conventions so that other programmers can easily read and modify your code. Professional software developers (almost!) always follow coding conventions of one form or another. Conventions may be specific to a programming language, or specific to a company or organisation (e.g. Apache).

## INTERRELATION OF CLASSES

### Inheritance

- A highly important kind of class relationship is *inheritance*.
- Inheritance means that a class (class B) is *derived* from another class (class A).
- Class A is called *superclass* (of class B), class B is called *subclass* (of class A).
- Class B *inherits* the members of class A

## INTERRELATION OF CLASSES

### Inheritance

- Optionally, class B can...
  - contain *additional* members which are not members of class A
  - *replace* (“override”) methods of class A.

# INTERRELATION OF CLASSES

## Inheritance

- Examples:
  - Superclass: `Bird`. Subclasses of `Bird`: `Flamingo`, `Eagle`, `Raven`
  - Superclass: `Person.` Subclasses of `Person`: `Student, Teacher`
  - Superclass: `GeometricFigure`. Subclasses of `GeometricFigure`: `Circle`, `Triangle`, `Rectangle`, `Ellipse`

## INTERRELATION OF CLASSES

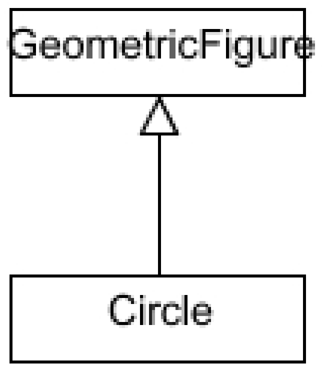
### Inheritance

- A class B which is a subclass of some class A can be superclass of another class C. E.g.,
    - Superclass: `GeometricFigure`
      Subclass of `GeometricFigure`: `Ellipse`
      Another subclass of `Ellipse`: `Circle`
    - Multiple classes related by inheritance form a special class hierarchy called *inheritance hierarchy*.
- Inheritance models the so-called (strong) *is-a relationship* among classes: an eagle *is a* bird, a student *is a* person…

# INTERRELATION OF CLASSES

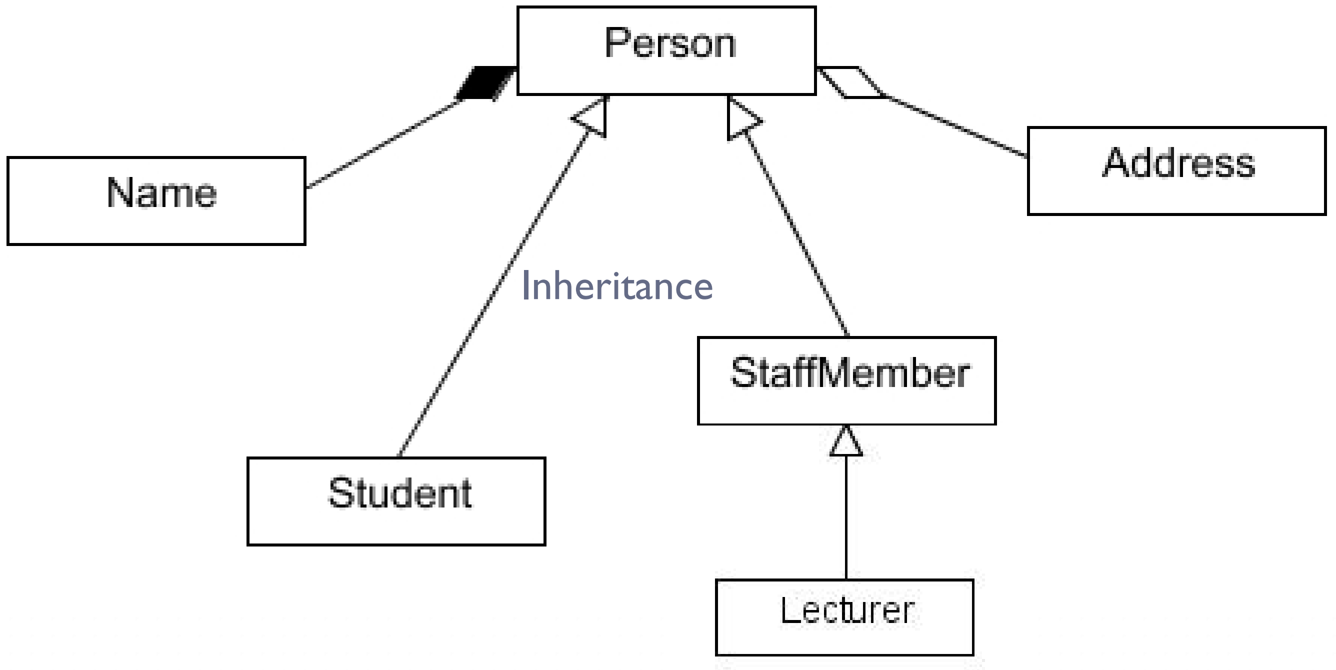
## Inheritance

Inheritance in UML:



## INTERRELATION OF CLASSES
### Inheritance

Another example in UML notation:



## INTERRELATION OF CLASSES

### Inheritance

- In Java, the keyword `extends` is used to specify inheritance.
- In the following example, `Circle` is the subclass and `GeometricFigure` is its superclass:

```
class Circle extends GeometricFigure {

   private double radius;

   public Circle(double radius) {
      this.radius = radius;
   }
   public double getArea() {
      return radius*radius*PI;
   }
   public void display() {
      System.out.println("This is a circle with radius " + radius);
   }
 }
```

## INTERRELATION OF CLASSES
### Inheritance

- And here is the code of the superclass:

```
class GeometricFigure {
   public static final double PI = 3.14159265;
   private boolean filled;

   public GeometricFigure() {
      filled = false;
   }
   public boolean isFilled() {
      return filled;
   }
   public void setFilled(boolean filled) {
      this.filled = filled;
   }
   public void display() {
      System.out.println("This is some geometric figure.");
   }
}
```

## INTERRELATION OF CLASSES

### Inheritance

- Each superclass can have multiple subclasses:

```
class Rectangle extends GeometricFigure {

   private double width;
   private double height;

   public Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
   }

   public double getArea() {
      return width*height;
   }
}
```

## INTERRELATION OF CLASSES

### Inheritance

- Inheritance in Java is *transitive**: if class X extends class Y, and class Y extends class Z, we say also that "X extends Z"
- If x is an instance (an object) of class X, it is also an instance (object) of class Y and Z
- Some terminological equivalences:
  "Class X *extends* class Y"
  <->
  "Class X *inherits* [properties, fields, members, methods...] from class Y"
  <->
  "Class X is *derived* from class Y"

  <->
  "Class Y *generalizes* class X"

- *Transitive definition from Lexico (Oxford): (of a relation) such that, if it applies between successive members of a sequence, it must also apply between any two members taken in order. For instance, if A is larger than B, and B is larger than C, then A is larger than C.

## INTERRELATION OF CLASSES
### Inheritance

- A subclass inherits all fields and methods from its superclass, except for constructors and private fields
- Subclasses can also declare *additional* methods and fields
- E.g., `PI`, `isFilled()` `setFilled()` and `display()` are members of `Circle`, as well as `radius`, `Circle()` and `getArea()`

## INTERRELATION OF CLASSES
### Inheritance

- Special case "private":
- Illegal within class `Circle`:

```
public void display() {
   System.out.println("This is a circle with radius " + radius);
   System.out.println("The circle is "+ (filled?"filled":"unfilled"));
}
```

- Legal (Why?):

```
public void display() {
   System.out.println("This is a circle with radius " + radius);
   System.out.println("The circle is "+
   (isFilled()?"filled":"unfilled"));
}
```

We are accessing the value of the private field `filled` using a "getter" type method, rather than trying to access it directly as per the example above

# INTERRELATION OF CLASSES

## Inheritance

- The `instanceof` operator provides a way to test whether an object has a certain class.

`instanceof` also works with superclasses:

```
Circle myCircle = new Circle(21.0);
System.out.println(myCircle instanceof Circle);
System.out.println(myCircle instanceof GeometricFigure);
```

yields

```
true
true
```

- However, your code should make use of `instanceof` only exceptionally (as it doesn't conform to the principles of static typing)

## INTERRELATION OF CLASSES

### Inheritance

- A subclass can not only declare members in addition to those inherited from the superclass, but it can also *override* methods of the superclass:

  If a non-static method of the subclass has *the same signature* as a non-static method of the superclass, then the method in the subclass *overrides* the method in the superclass.

  ("Signature" of a method means: method name and the number and <u>types</u> of the parameters)

## INTERRELATION OF CLASSES

### Inheritance

- An overriding method can return an object of the same class as the object returned by the overridden method,  or an object of a subclass of that class
- Overriding methods should be annotated with `@Override`

## INTERRELATION OF CLASSES

### Inheritance

- When a method of the subclass overrides a method of the superclass, the subclass method effectively *replaces* the method in its superclass if used by objects of the subclass.
- Assume X is a subclass of Y and in class Y there is a method `m(...)`
- Further assume that class X contains a method `m(...)` which overrides method `m` in the superclass Y
- If some object x is an instance of class X, the method call `x.m(...)` refers to the method `m(...)` in class X.
- If some object y is an instance of class Y but *not* of class X, the method class `y.m(...)` refers to the method `m(...)` in class Y

## INTERRELATION OF CLASSES
### Inheritance

- Overriding (and overloading) methods is the most important variant of a principle called *polymorphism*
- Details about polymorphism in subsequent lectures

- Examples…

- In the previous example code, method `display()` in `Circle` overrides method `display()` of `GeometricFigure`. In contrast, the code in `Rectangle` does not override any method.

## INTERRELATION OF CLASSES

### Inheritance

```
Circle myCircle = new Circle(11.8);
myCircle.display();
```

prints `This is a circle with radius 11.8`

Whereas

```
GeometricFigure gf = new GeometricFigure();
gf.display();
```

prints `"This is some geometric figure."`

And

```
Rectangle myRectangle = new Rectangle(8.32, 3.4);
myRectangle.display();
```

also prints `This is some geometric figure.`

## INTERRELATION OF CLASSES

### Inheritance

- <u>Constraint 1</u>: a method cannot be overridden with a method with more restrictive accessibility.
E.g., it is not possible to override a public method in the superclass with a private method in the derived class.

# INTERRELATION OF CLASSES

## Inheritance

- Constraint 2: a static method cannot be overridden.

If the subclass defines a static method with the same signature as a static method of the superclass, the two methods coexist. Which method is called depends on the compile-time type of the object, irrespectively of the "real" (i.e., runtime) type of the object. This is different from overriding!

## INTERRELATION OF CLASSES
### Inheritance

- Constraint 3: technically, a method cannot be overridden if it is not accessible outside of the superclass.
- E.g., it is not possible to override a private method
- If a method in the subclass has the same signature as a private method in the superclass, these two methods are unrelated and coexist.

Analogously to static methods, the compiler(!) decides which method to call (this is again different from overriding…!)

# INTERRELATION OF CLASSES

## Inheritance

**Example (1):**

```
class Base {
   void foo() { System.out.println("foo() of class Base"); }
}

class Child extends Base {
   void foo() { System.out.println("foo() of class Child"); }
}

Base someBase = new Base();
Child someChild = new Child();
Base someOtherChild = new Child();

someBase.foo();  // prints "foo() of class Base"
someChild.foo();  // prints "foo() of class Child"
someOtherChild.foo();  // prints "foo() of class Child"
```

## INTERRELATION OF CLASSES

### Inheritance

**Example (2):**

```
class Base {
   static void foo() { System.out.println("foo() of class Base"); }
}

class Child extends Base {
   static void foo() { System.out.println("foo() of class Child"); }
}

Base someBase = new Base();
Child someChild = new Child();
Base someOtherChild = new Child();

someBase.foo();  // prints "foo() of class Base"
someChild.foo();  // prints "foo() of class Child"
someOtherChild.foo();  // prints "foo() of class Base"!
```

## INTERRELATION OF CLASSES

### Inheritance

- We can disallow the overriding of a certain method explicitly using modifier `final`:

```
public final void display() {
  System.out.println("This is some geometric figure.");
}
```

- With this declaration within the superclass, the declaration of method `display` within class `Circle` would become illegal.

## INTERRELATION OF CLASSES

### Inheritance

- We have the possibility to invoke an overridden method by using the keyword `super`... if otherwise the overriding method would be called…

# INTERRELATION OF CLASSES

## Inheritance

- Using `super`, we could e.g. redefine `Circle's` display method as

```
public void display() {
    super.display(); // calls the method of the superclass
    System.out.println("This is a circle with radius " +
    radius);
}
```

Now, `myCircle.display()` would print both

```
This is some geometric figure.
This is a circle with radius 11.8
```

- `super` can be used in other methods too, not just in overridding methods.

## INTERRELATION OF CLASSES

### Inheritance

- Remember method *overloading?*
What again is the difference between method *overloading* and method *overriding* …?
- Overloading means there are multiple methods with the **same name but a different number of parameters or different parameter types** (i.e., different signature). These methods simply coexist. Which one is invoked is decided by the compiler, simply by looking at a method call's arguments.
- In contrast to that, overriding is resolved partially at runtime, using a mechanism called *dynamic dispatch* (-> next lecture)

# INTERRELATION OF CLASSES

## Inheritance

Example:

- In superclass: `myMethod(int x, double y)`
+ in subclass: `myMethod(int x, double y)`
➔ overriding

- In superclass `myMethod(int x, double y)`
+ in subclass or superclass: `myMethod(float x)`
➔ overloading

## INTERRELATION OF CLASSES

### Inheritance

- Overriding works only with methods. It is not possible to “override” a field.
- However, you can declare a field in the subclass which has the same name and type as a field in the superclass.

(Doesn't mean you should!)

- Both fields then "co-exist" in an object of the subclass (with potentially different values), but within the subclass the field declared by the subclass <u>shadows</u> the field declared by the superclass (with no runtime resolution). However, the superclass field can also be accessed, again using `super.fieldname` (if not declared private in the superclass).

## INTERRELATION OF CLASSES

### Inheritance

- Within methods declared in the superclass, the superclass version of field is used, even if the method which accesses the field is inherited by the subclass and used on subclass objects.
- Such “shadowing” of fields is rarely done, because it can easily lead to confusion.

## INTERRELATION OF CLASSES

### Inheritance

- Constructors play a special role w.r.t. inheritance
- In contrast to ordinary methods they are <u>not</u> inherited by subclasses
- In a constructor of the subclass, you can invoke a constructor of the superclass using `super(...)`
- Any constructor of the subclass invokes firstly a constructor of the superclass
- If an explicit superclass-constructor call is missing, Java inserts a call `super()` at the beginning of your subclass constructor. (This leads to an error in case no zero-parameter constructor exists in the superclass.)

## INTERRELATION OF CLASSES
### Inheritance

- In the examples before, `Circle` declares the constructor

```
public Circle(double radius) {
   this.radius = radius;
}
```

Since a call of a superclass constructor is missing here, Java implicitly adds one:

```
public Circle(double radius) {
   super();  // added automatically
   this.radius = radius;
}
```

`super()` invokes the following constructor of `GeometricFigure`:

```
public GeometricFigure() {
   filled = false;
}
```

## INTERRELATION OF CLASSES
### Inheritance

- `super(…)` can also be used to invoke constructors of the superclass which have parameters:

```
public ClassName(…) {
   super(12, true);
   …
}
```

- It is not possible to invoke the superclass constructor using the name of the constructor

```
public Circle(double radius) {
   GeometricFigure(); // illegal
   this.radius = radius;
}
```

## INTERRELATION OF CLASSES
### Inheritance

- Each time an instance of a class is created, the super-super-...-superclass constructors along the inheritance hierarchy are executed
- E.g., if a class X has a superclass which has itself a superclass, the constructor of the latter is processed first, then the direct superclass constructor, and finally the constructor of class X
- This is called *constructor chaining*

## INTERRELATION OF CLASSES

### Inheritance

- In some OOP languages such as C++, it is possible that a certain class is derived from several classes or with the help of interface-like concepts, such as *traits* (e.g., Scala) -> *multiple inheritance*.
- But because this brings certain subtle difficulties with it, this is not allowed in Java. Java supports *single inheritance* only.
- However, this applies only to classes, not to *interfaces* (more on this in the next lecture).

## INTERRELATION OF CLASSES

### Inheritance

- More details about polymorphism, overriding, etc in the next lecture