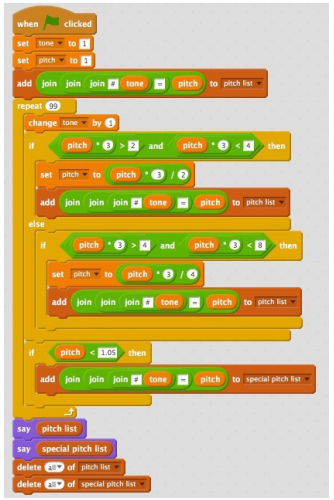
Preliminaries: Road to Object-Oriented Programming and Design (OOPL/D)

Brief History

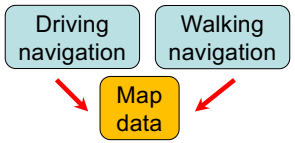
- In good, old days… programming languages had no notion of structures (or modularity)
  - One dimensional code was sufficient to satisfy most requirements.
    - Written and maintained from the first to the last line on a line-by-line basis.



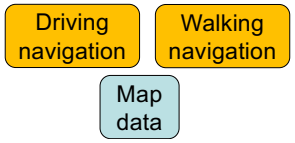
- As the size and complexity of software increased, languages required "complexity management mechanisms."
  - "Go to" statements to control program flows.
    - Produced a lot of "spaghetti" code
      - "Go to" statements considered harmful.
  - Modules
    - Module: A chunk of code
    - Modularity: Making modules self-contained and independent from others

Modularity

- Making modules self-contained and independent from others
- Goals: Improve productivity and maintainability
  - Improve productivity through higher reusability
    - Can reduce production costs



- Making modules self-contained and independent from others
- Goals: Improve productivity and maintainability
  - Improve maintainability through clearer separation of concerns
    - Changes can be made in a module without changing others.
    - Bugs can be fixed in a module without changing (or worrying about) others.
    - The scope of work can be narrowed.
  - Can reduce maintenance (incl. code revision) costs

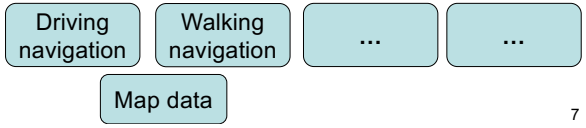


Modules in SPLs and OOPLs

- Modules in Structured Prog. Languages (SPLs)
  - Structure = a set of variables
  - Function = a block of code
- Modules in Object-Oriented PLs (OOPLs)
  - Class = a set of variables (data fields) and functions (methods)
  - Interface = a set of functions (methods)
- Key question: How to take advantage of modules to address reusability and separation of concerns?
  - how to define modules?
  - how to separate a module from others?
  - how to let modules interact with each other?

OOPLs

- Intended to seek coarse-grained modularity.
  - The size of each module is often bigger in OOPLs (than in SPLs).
- Designed with extensibility in mind to enhance productivity and maintainability further.
  - How easy (cost effective) to add new modules to implement new/modified requirements.
  - How to make software more flexible/robust against changes in the future.
- Key question: How to take advantage of modules to address reusability, separation of concerns and extensibility?



Goal of CS680

- Learning about ***design*** and ***organization*** of object-oriented programs.
  - Learning how to divide single-function or single-class code to multiple classes and interfaces.
  - Will NOT cover/explain
    - Why single-function and single-class code are not good when you have large-scale and complex requirements.
    - How to divide single-function code to multiple functions.

Looking Ahead

- OOPLs are good for modularity, but not perfect.
- OOPLs still have some room for improvements.
  - Hard to modularize cross-cutting concerns.
    - e.g. logging, security, error handling, DB access, etc.
    - Aspect Oriented Programming and dependency injection
  - Highly modular (and expressive) code often look redundant (or repetitive).
    - Functional programming
      - Makes code less redundant/repetitive.
    - Lambda expressions in Java
      - Intended to make modular OO code less redundant/repetitive.

Encapsulation

Modularity Mechanisms in Java

- Class, interface and package
- Just using these mechanisms DOES NOT automatically mean that your code will modular.
- You need to use them well (in well-thought-out ways).
  - One of them is to follow the encapsulation principle.

What is Encapsulation?

- Hiding each class's internal details from its clients
  - To make classes modular (or blackbox).
  - "Client" ("client code"):
    - A piece of code that uses a class in question.
- Things to do:
  - Always make your data fields private or protected.
  - Make your methods private or protected as often as possible.
  - Avoid public accessor methods (getter/setter methods) whenever possible.
  - Make your classes "final" whenever appropriate.

Example

```
final public class Person{
    private int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }

Person person = new Person(123456789);
int ssn = person.getSSN();
…
```



```
final public class Person{
    private int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }

Person person = new Person(123456789);
int ssn = person.getSSN();
…
```

- What if you have a runtime error about a person's SSN? (e.g., The SSN is wrong or null.) Where is the source of the error, inside or outside `Person`?
  - You can tell it should be outside `Person` for sure.
    - A bug(s) should exist before calling `Person`'s constructor or after calling `getSSN()`.
  - You can narrow the scope of your debugging effort.
    - You can be more confident about your debugging.

Recap

- Specify visibility for every data field and every method.
- Do not to skip specifying it; Do not use the default (package-private) visibility.
- It is always important to be aware of the visibility of each data field and method.

Violation of Encapsulation

- If `Person` looks like these, you cannot be so sure about where to find a bug.

```
final public class Person{
    private int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; }
    public setSSN(int ssn){ this.ssn = ssn; } }
```

```
final public class Person{
    public int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }
```

- If `Person` looks like this, you cannot be so sure about where to find a bug.
  -
```
final public class Person{
    private int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public String getSSN(){ return this.ssn; }
    public setSSN(int ssn){ this.ssn = ssn; } }
```
  -
```
Person person = new Person(123456789);
int ssn = person.getSSN();
……
person.setSSN(987654321);
```
  - You or your team mates may write this code by accident.
    - It may look stupid, but it is not that uncommon in reality.
  - Don't define public setter methods if you don't need them.

- There are a good number of data structures that don't have to be modified once they are initialized.
  - e.g., globally-unique IDs (GUIDs), customer IDs, product IDs, etc.
- Define them as private/protected data fields.
- No need to define setter methods.

Violation of Encapsulation (cont'd)

- Supose you are the provider of `Person` (or API designer for `Person`)
  - Your teammates will use your class for *their* work.

```
public class Person{
    protected int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }
```


- Your class will allow your teammates to mess up SSNs.

```
public class Person{
    protected int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }

public class MyPerson extends Person{
    MyPerson(int ssn){ super(ssn); }
    public void setSSN(int ssn){ this.ssn = ssn; } }
```

- To avoid this,
  - Replace "protected" with "private."
  - Turn the class to be "final."

In a Modern Software Dev Project…

- No single engineer can read, understand and remember the entire code base.
- Every engineer faces time pressure.
- Any smart engineers can make unbelievable errors VERY EASILY under time pressure.
- Your code should be *preventive* for potential misuses and errors.

Be Preventive!

- Encapsulation
  - looks very trivial.
  - is not that important in small-scale (toy) software
    - because you can always manage (i.e., read, understand and remember) every aspect of the code base.
  - is very important in larger-scale (real-world) software
    - because you cannot always manage (i.e., read, understand and remember) every aspect of the code base.

Example Cases

```
public class Sensor{
    protected int id;
    Sensor(...){ this.id = ... }
    public int getId(){ return this.id; } }
```

```
public class SensorV2 extends Sensor{
    Person(int id){ super(id); }
    public void setId(...){ this.id = ... } }
```



```
public class Package{
    protected int id;
    Package(int id){ this.id = id; }
    public int getId(){ return this.id; } }
```

```
public class TrackablePackage extends Package{
    private String trackingId;
    TrackablePackage(int id, String trackingId){
        super(id);
        this.trackingId = trackingId; }
    public String getTrackingId(){ ... }
    public void setTrackingId(String trackingId){ ... } }
```

```
- public class MissionComponent{
      private double refDistance; // in the imperial system
      MissionComponent(...){ refDistance = ...; }
      public double getRefDistance(){ return refDistance; } }

- public class MissionComponent{
      private double refDistance; // in the imperial system
      private double refDistanceMetric;
      MissionComponent(...){refDistance = ...;
                            refDistanceMetric = ...}
      public double getRefDistance(){ return distance; }
      public double getRefDistanceMetric(){...}
      public double setRefDistanceMetric(){...}}
```

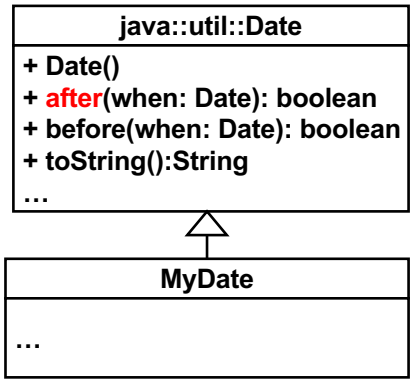

Sounds Easy to Do?

```
final public class Person{
    private int ssn;
    Person(int ssn){ this.ssn = ssn; }
    public int getSSN(){ return this.ssn; } }
```

- Once you finish up writing these 4 lines, wouldn't you write a setter method automatically (i.e. without thinking about it carefully)?
  - "I always define both getter and setter methods for a data field. I can delete unnecessary ones anytime later."
  - "Well, let's define a setter just in case."
  - Think twice. Fight that temptation.
    - Define the setter method only when you absolutely need it.

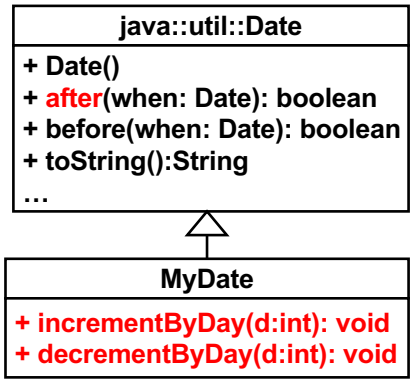
Inheritance (Generalization)

An Example of Inheritance




```
Date d = new Date();
d.after( new Date() );

MyDate md = new MyDate();
md.after( d );
// after() is inherited to MyDate
```

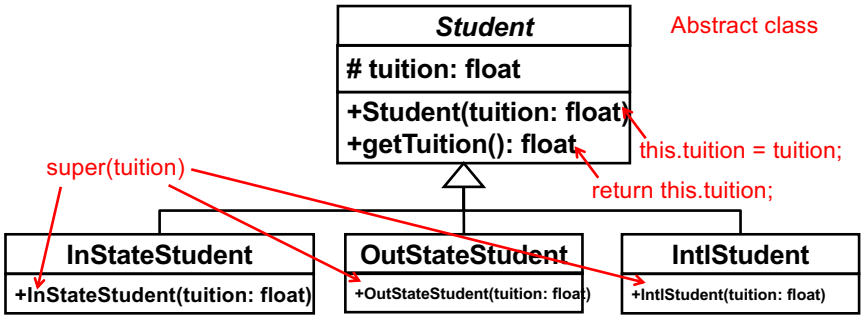
- Generalization-specialization relationship
  - a.k.a. "is-a" relationship; MyDate is a (kind of) Date.
- A subclass can *inherit* all public/protected data fields and methods from its super class.
  - Exception: Constructors are not inherited.




```
Date d = new Date();
d.after( new Date() );

MyDate md = new MyDate();
md.after( new Date() );

d.incrementByDay(1);
// Compilation error
md.incrementByDay(1);    // OK

d = new MyDate();        // OK
d.incrementByDay(1);
// Compilation error
(MyDate)d.incrementByDay(1); // OK
```

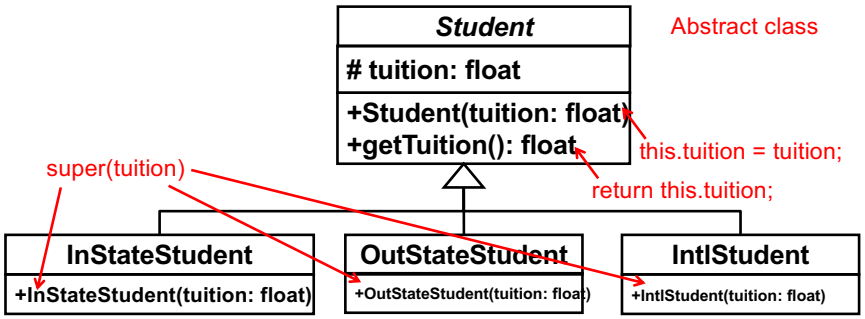
- A subclass can *extend* its super class by adding extra data fields and methods.
- An instance of a subclass can be assigned to a variable typed with the class's superclass.

Another Inheritance Example




```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStudent(3000) );

Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

- What are printed out in the standard output?



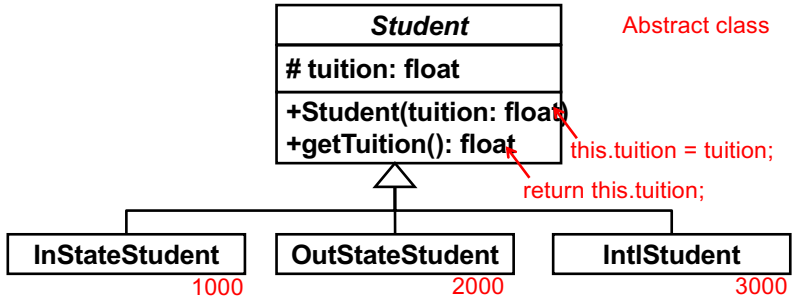

```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStudent(3000) );

Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

- 2000
  1000
  3000

Exercise (Not HW)

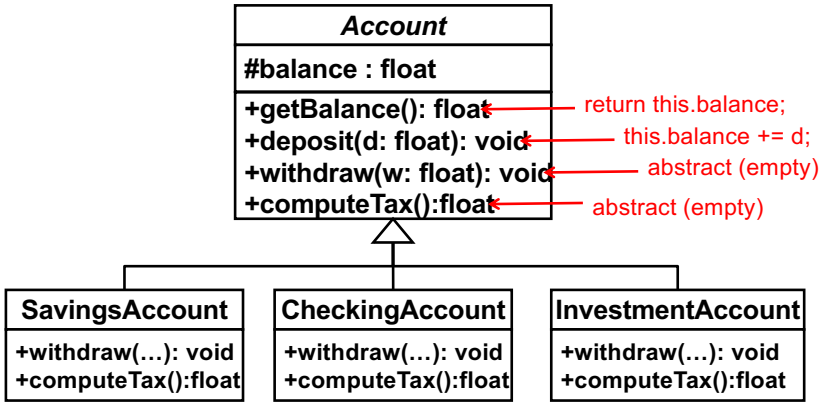
- Learn about collections (e.g. ArrayList) and generics in Java.
- Learn how to use `java.util.Iterator`.
- This code runs.
  - `ArrayList<Student> al = new ArrayList<Student>();`
    `al.add( new OutStateStudent(2000) );`
    `System.out.println( al.get(0).getTuition() );` → 2000
- This one doesn't due to a compilation error.
  - `ArrayList al = new ArrayList();`
    `al.add( new OutStateStudent(2000) );`
    `System.out.println( al.get(0).getTuition() );`
- Understand what the error is and why you encounter the error.

Polymorphism

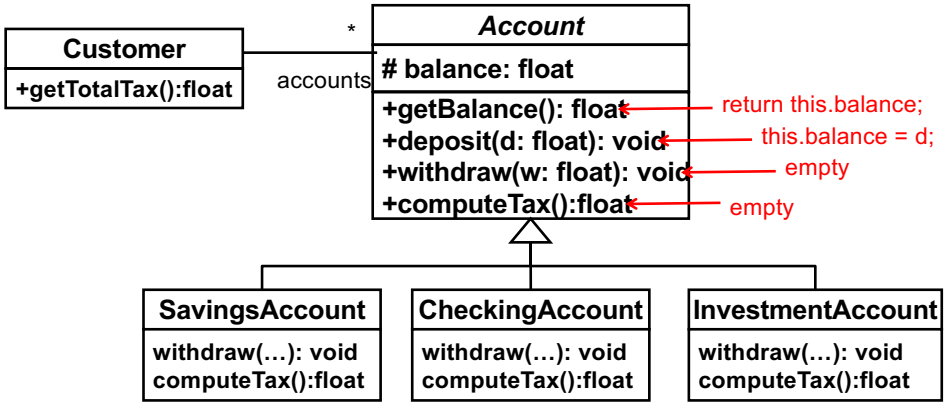



```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStateStudent(3000) );

Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

- All slots in "students" (an array list) are typed with Student (super class).
- Actual elements in "students" can be instances of Student's subclasses.



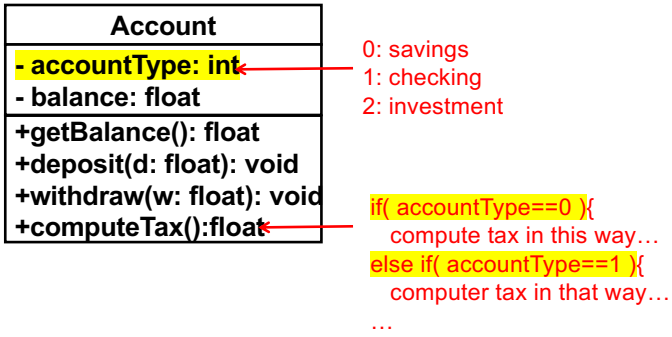
- Subclasses can redefine (or override) inherited methods.
  - A savings account may allow a negative balance with some penalty charge.
  - A checking account may allow a negative balance if the customer's savings account maintains enough balance.
  - An investment account may not allow withdrawals.




```
public float getTotalTax(){
     float totalTax;
     Iterator<Account> it = accounts.iterator();
     while( it.hasNext() )
            totalTax += it.next().computeTax() );
     return totalTax; }
```

- Polymorphism can effectively eliminate conditionals.
  - Conditional statements are a VERY typical source of bugs.

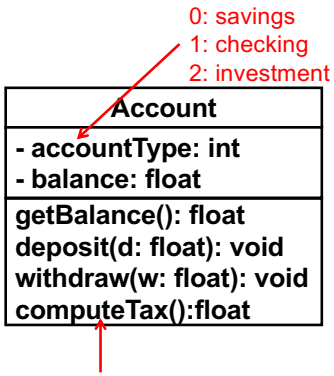
If Inheritance and Polymorphism are not Available…



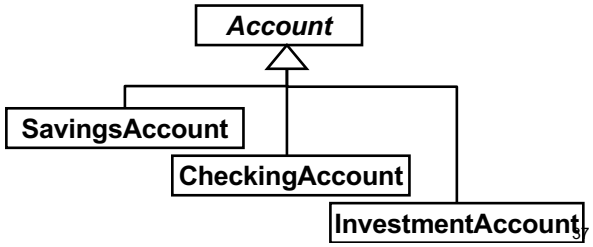
- Issues: ***magic numbers*** (a.k.a. type code) and ***conditionals***

What's Wrong with Magic Numbers

- They often scatter in code and get harder to maintain.

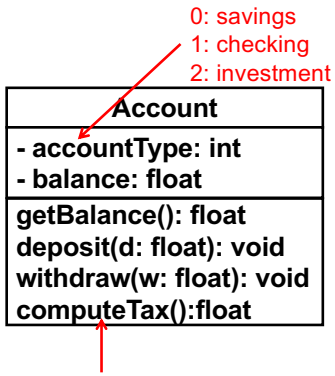


- When you add/remove magic numbers, you have to find all the places where they are used and revise those places.
- Serious maintainability issue if you use many magic numbers and if you often add/remove magic numbers.

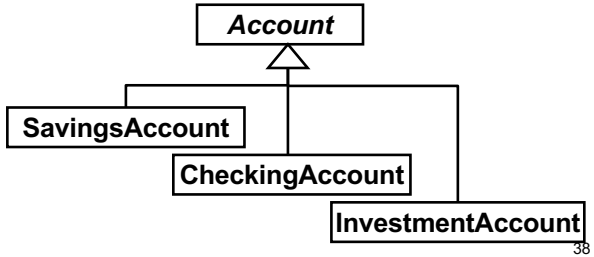


What's Wrong with Conditionals?

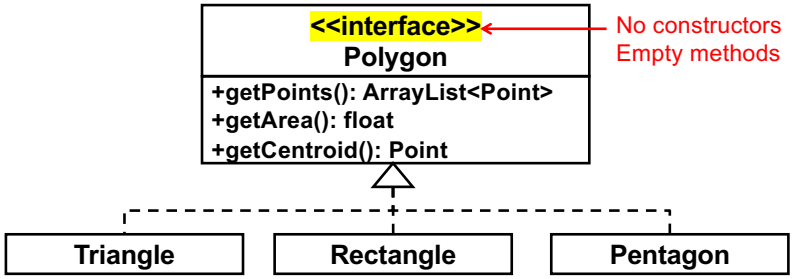
- They often scatter in code and get harder to maintain.



- Conditionals are a common source of bugs
  - Especially when there are many possible conditions and conditional branches are structured in complex manners.



Another Polymorphism Example




```
ArrayList<Polygon> p = new ArrayList<>();
p.add( new Triangle( new Point(0,0),
                     new Point(2,2),
                     new Point(1,3) ));
p.add( new Rectangle ( new Point(0,0)... ));
Iterator<Polygon> it = p.iterator();
while( it.hasNext() ){
  Polygon nextPoly = it.next();
  System.out.println( nextPoly.getPoints() );
  System.out.println( nextPoly.getArea() );
  System.out.println( nextPoly.getCentroid() ); }
```


Exercise (Not HW)

- Write and run some code for the Student, Account and/or Polygon examples
  - If you are not familiar with class inheritance and polymorphism.
  - Understand other program elements in Java, such as final, static, super, and constructors.
  - Understand how to use the Date and Time API.
    - See subsequent slides.

Date and Time API in Java

- `java.util.Date` (since JDK 1.0)
  - Poorly designed: Never try to use this class
    - It still exists only for backward compatibility
- `java.util.Calendar` (since JDK 1.1)
  - Deprecated many methods of `java.util.Date`
  - Limited capability: Try not to use this class
- Date and Time API (`java.time`)
  - Since JDK 1.8
  - Always try to use this API.

Date and Time API: `Instant`

- Represents an instantaneous point on the timeline, which starts at 01/01/1970 (on the prime Greenwich meridian).
  - Can be used as a timestamp.
- **`Duration`**
  - Represents an amount of time in between two `InstantS`

```
Instant start = Instant.now();
...
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long timeElapsedMSec = timeElapsed.toMillis();
```

Date and Time API: "Local" Classes

- **`LocalDate, LocalTime, LocalDateTime`**
  - Used to represent date and time without a time zone (time difference)
  - Apply leap-year rules automatically.

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(2009, 9, 10);
LocalDate 18thBirthday = birthday.plusYears(18);
birthday.getDayOfWeek().getValue();
```

- **`Period`**
  - Represents an amount of time in between two local date/time.

```
Period period = today.until( 18thBirthday );
period.getDays();
```

Date and Time API: Other Classes

- **`TemporalAdjusters`**
  - Utility class (i.e., a set of static methods) that implements various calendaring operations.
    - e.g., Getting the first Sunday of the month.
- **`ZonedDateTime`**
  - Similar to `LocalDateTime`, but considers time zones (time difference) and time-zone rules such as daylight savings.
- **`DateTimeFormatter`**
  - Useful to parse and print date-time objects.