Advanced Object-Oriented Programming

Student Workbook

Version 1.1

Table of Contents

Module 1

Interfaces

Section 1–1

What is an Interface

What is an Interface?

- **Interface - Noun**
  - A point where 2 systems, subjects, organizations meet and interact
  - A device where a person can interact with a computer
  - An adapter that allows you to connect 2 devices together
  - Examples
    - The software has a simple and intuitive interface that makes it easy for users to navigate
    - The team designed a hardware interface so that they could connect 2 machines and share data between them

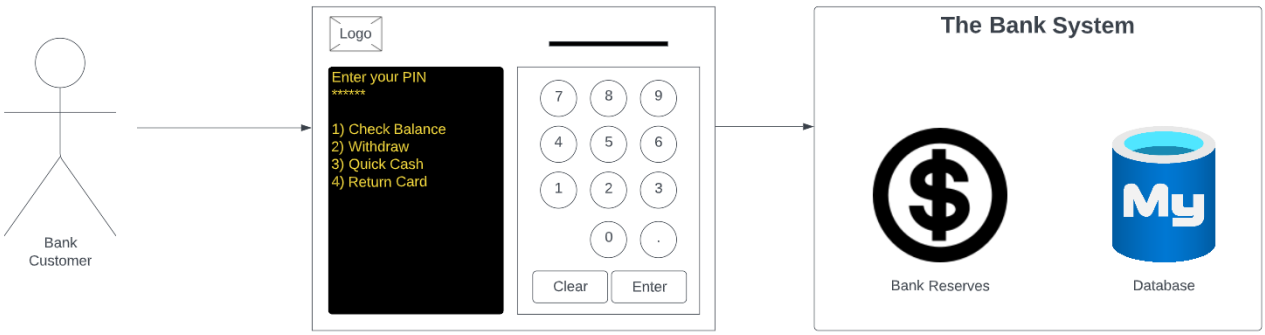
- **Interface - Verb**
  - To interact with another person or system
    - HR needs to interface with the IT team to ensure that all candidates have the correct qualifications
    - The Xbox controller allows players to interface with the game to control their character
    - The new software will interface with the database for a seamless data entry process

- **Both the noun and the verb definitions of Interface are applicable in Software Engineering**
  - We design and build User Interfaces (noun) so that users and interface (verb) with our software
  - We design and build Interfaces (noun) so that one software system can interface (verb) with another
- **In OOP the public methods of a class define that class's interface**
- **All public methods of a class allow outside code to *interface* with the class/object**
- **Java also introduced an `interface` keyword and construct**
  - In the next few modules we will learn about the `interface` construct and how we use interfaces to write better software

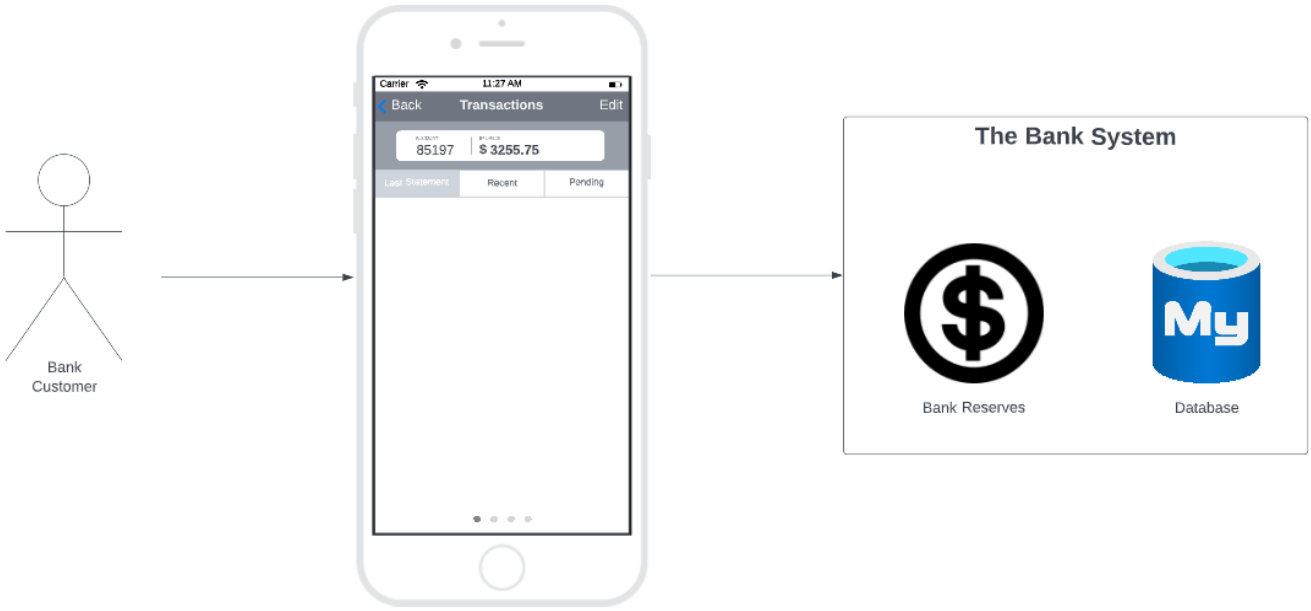
User Interfaces

- **In this course our focus is NOT User Interfaces, but they are useful in helping us to understand Interfaces in general**

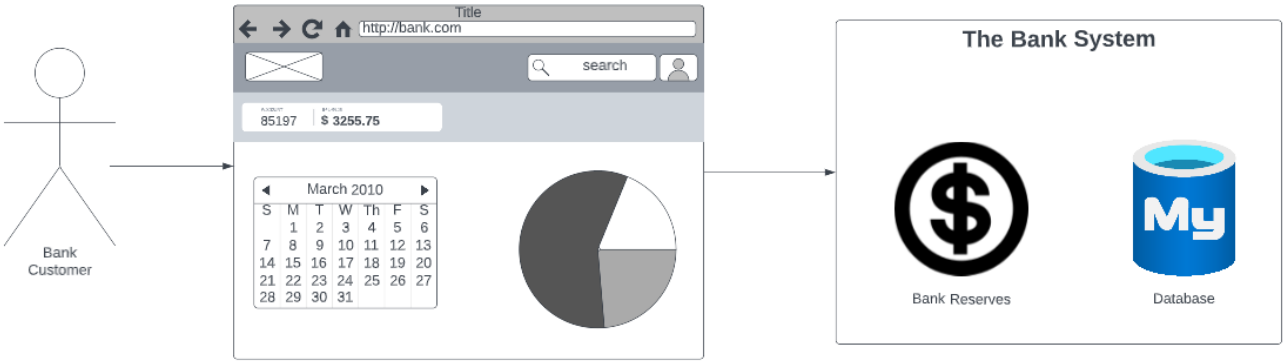
- **ATM**
  - A bank ATM gives the user the ability to access their bank accounts
  - Note that the Interface does not actually DO anything
    - It is only a doorway to the Bank Account system that does the actual work



- **Mobile Application**
  - A mobile application is simply a different interface to the same banking system
  - The Interface may provide different functionality, but the system behind the scenes is the same system



- **A web application is also another interface into the same banking system**
  - This will give the user different capabilities in actions that they can perform at the ATM or on their mobile devices



Interfaces

- **Not all interfaces are designed for users**
- **An interface defines how one entity can interact with another**
  - The interface does not know the inner workings of an object
  - The interface only defines what functions will publicly be available

Section 1–2

Abstraction through Interfaces

Abstraction Review

- **Abstraction is the process of simplifying your code to only expose what is necessary**
  - The User Interface of an ATM does NOT give the user access to all bank account functionality
    * They cannot open a new bank account
    * They cannot apply for a new loan
  - We try to keep the interface simple, by not showing the user options which they cannot perform at an ATM

- **Universal remote control vs Basic remote control**
  - Do you need ALL of the buttons - or just channel and volume up/down?

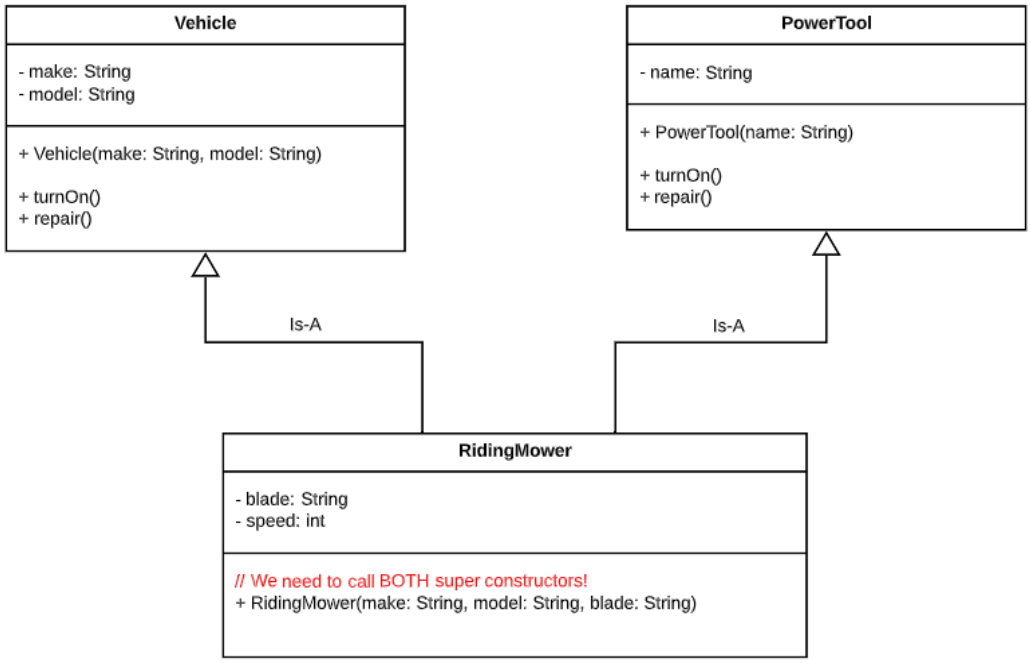
Abstract Class Review

- **Abstract classes cannot be instantiated**
- **Abstract classes can have abstract methods**
  - Abstract methods have no method body and must be overridden by the child class
- **Some programming languages allow extending multiple classes**
  - C++ and Python to name a couple
  - The idea is that you may need the functionality of 2 different classes when you create your new class
- **Java DOES NOT allow multiple inheritance**
  - Other modern languages like C# also do not allow for multiple inheritance

Why NOT use Multiple Inheritance?

Example

This example is for a small repair shop that can repair anything with a motor. A riding mower is both a vehicle and a power tool. With multiple inheritance we might have the `RidingMower` class extend both `Vehicle` and `PowerTool`.

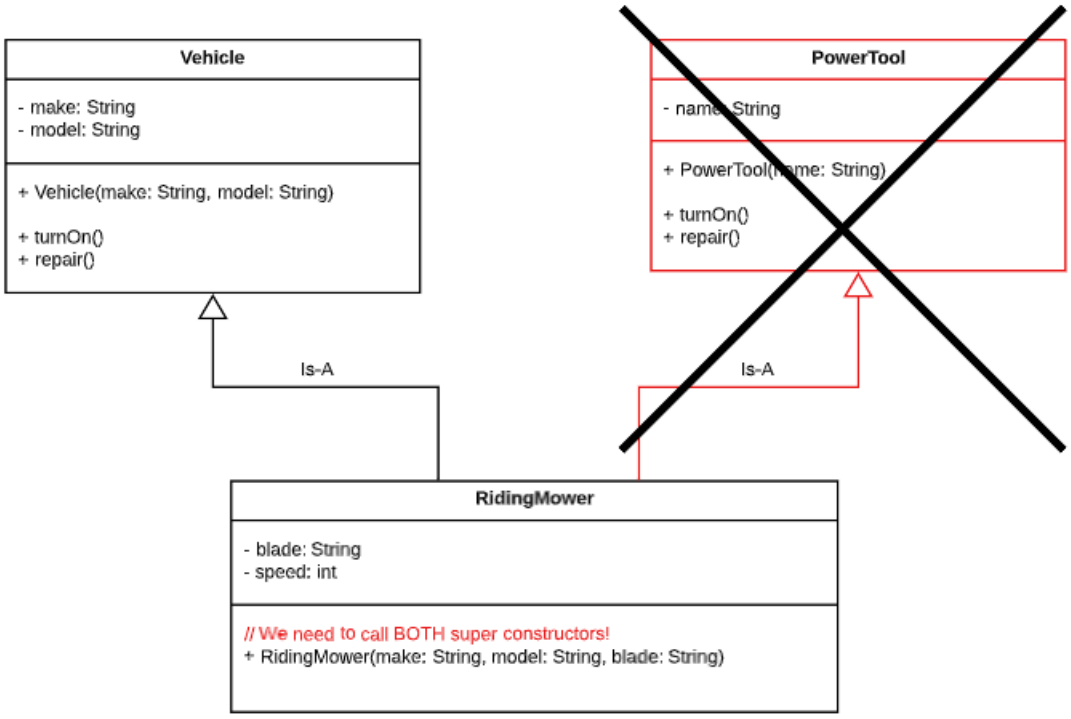


- **Multiple Inheritance causes potential problems**
  - Both the `Vehicle` and `PowerTool` constructors must be called when creating a `RidingMower`
  - When we execute the following code - which `repair` method is executed, the `Vehicle` or the `PowerTool`?

```
RidingMower mower = new RidingMower("Toro","S3500","35 in");
mower.repair(); // which super class is used here?
```

- **Developers had to ensure that the correct code was going to be executed**
  - This often led to fragile code

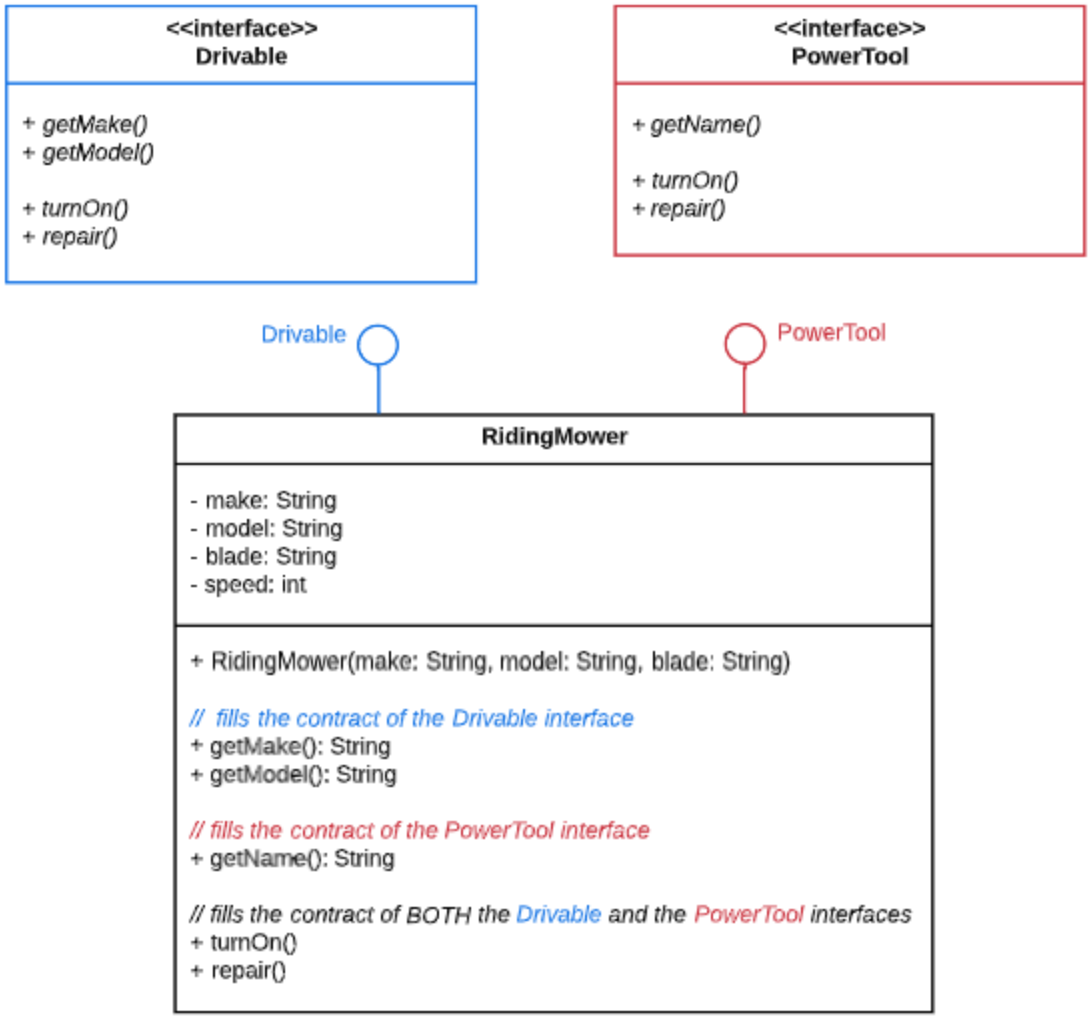
- **The Java team decided that Multiple Inheritance would NOT be allowed in Java**
  - In Java you can extend only a **single parent** class in Java



Introduction to Interfaces

- **Developers still need the ability to implement the functionality of multiple datatypes**
  - Inheriting properties and methods from multiple classes is still a good idea
- **Java introduced the concept of an `interface` construct to allow for multiple inheritance**
- **An `interface` is like a completely `abstract` class**
  - **All methods** are **abstract** by definition
  - **All methods** must be **public**
  - There can be **NO implementation code** within the interface
- **Classes `implement` interfaces**
  - This is similar to extending an abstract class
- **When a class implements an interface, it enters a contract that it will implement and expose the public methods defined by the interface**
  - A class **MUST** override **ALL** of the methods defined in the interface
- **A class can implement as many interfaces as necessary**

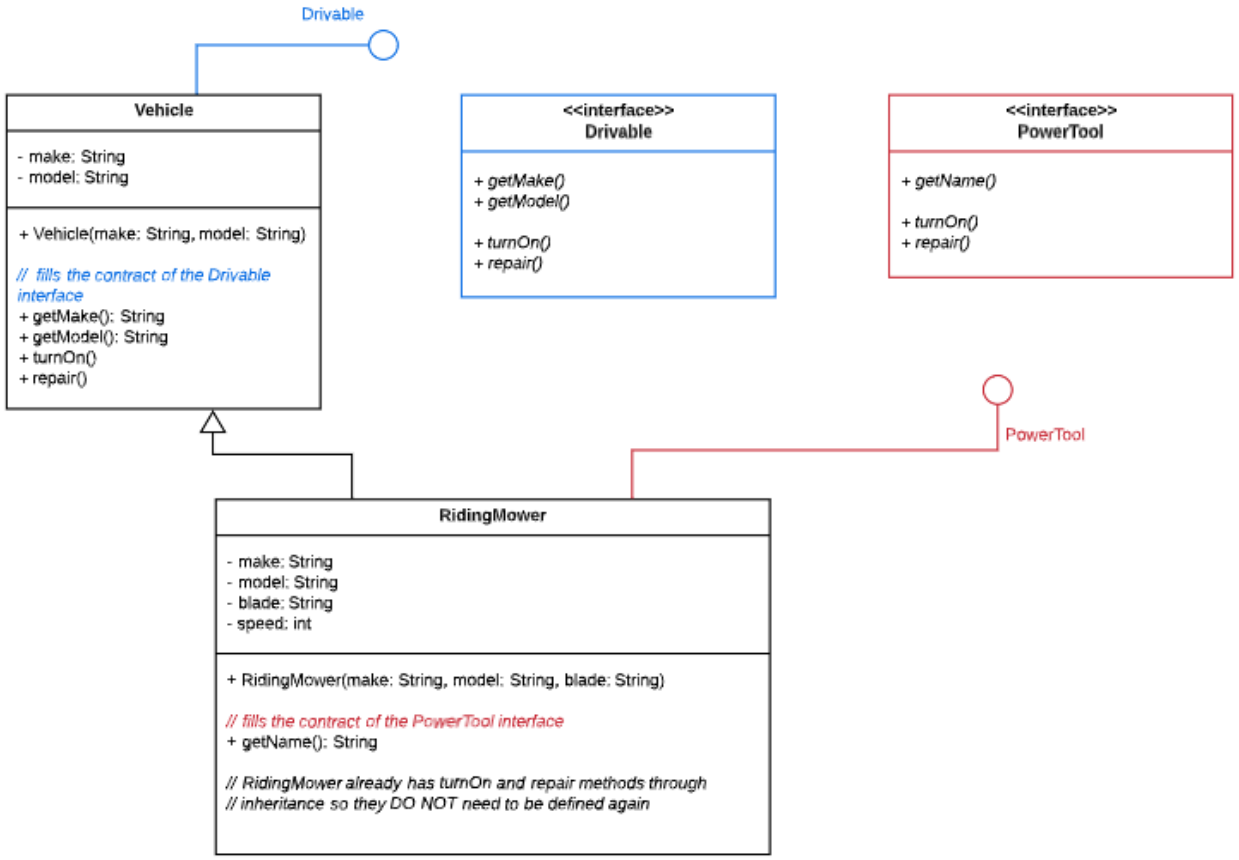
Introduction to Interfaces *cont'd*



- **Since an interface cannot have implementation code, it will never be ambiguous which method will be executed**
  - `mower.repair()` will **always** be executed in the RidingMower class

- **A class can still extend just one parent class, but it can also implement multiple interfaces**

- **If a parent class implements in interface all child classes will inherit the methods defined by that interface**



Exercise

Create a new folder in the `java-developer` directory and name it `workbook-6`. All of this week's exercises will be done in that folder.

EXERCISE 1

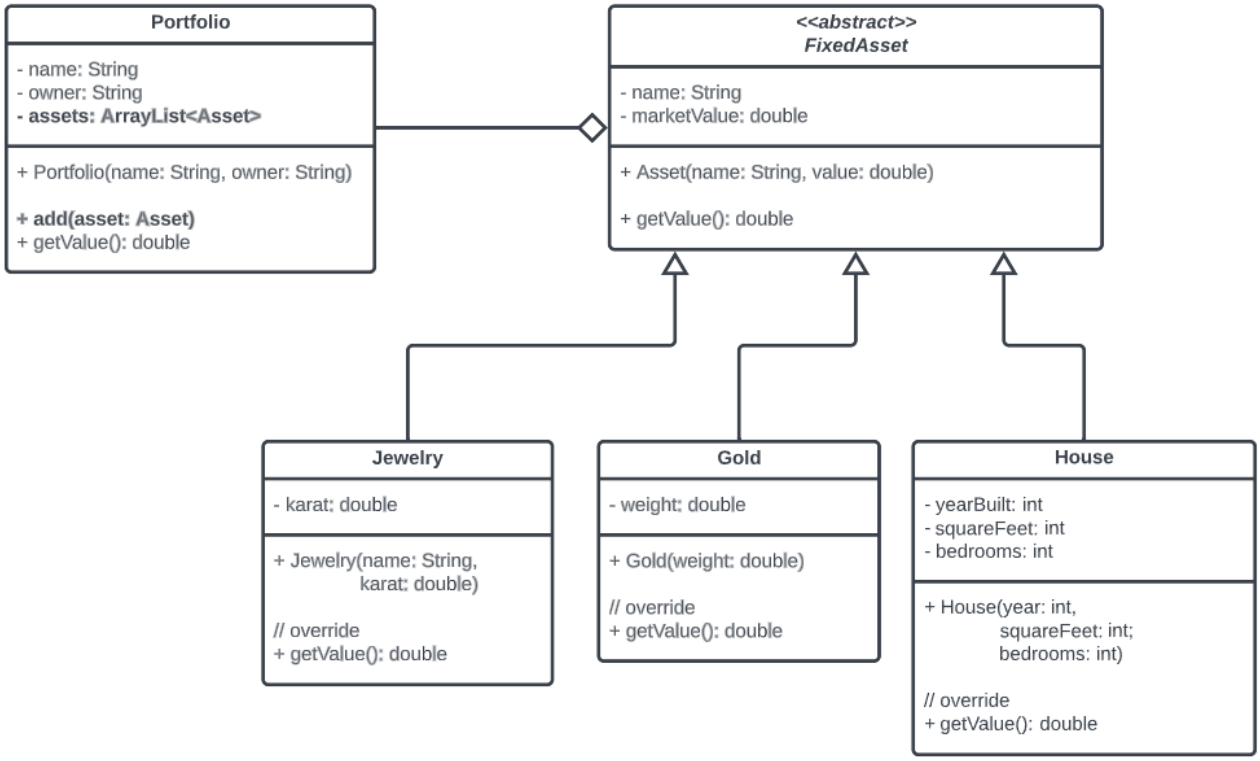
In this exercise work together in a small group to modify a class diagram for an asset portfolio application. A portfolio is made up of one or more `FixedAssets`.

Use a diagramming tool of your choice

drawio.com (free)
lucidchart.com (free version)
visual-paradigm.com (free version)
figma.com (free version)

Currently the portfolio is only made up of fixed assets such as jewely or houses.

The application is able to calculate the total value of all assets in the portfolio.



Part 1

We now also want the ability to include our more liquid accounts into our portfolio so that we can have a better estimation of what our actual net worth is.

This means that you should include a `BankAccount` class that has both `withdraw` and `deposit` methods. We also need to include debt.

A `CreditCard` class would include `charge` and `pay`.

NOTE: Liquid accounts are different than assets, because the value of `BankAccount` and `CreditCard` will change frequently, whereas fixed asset values do not. Fixed assets should NOT include `withdraw`, `deposit`, `charge` or `pay` methods. The `BankAccount` and `CreditCard` classes should not extend the `Asset` class.

You will need to create a new interface named `Valuable`. Ensure that the `Valuable` interface defines the `getValue()` method.

Also, consider what changes you will need to make to your `Portfolio` class so that your list of valuables includes your new classes.

Part 2

Create a folder in the `workbook-6` directory named `assets-diagrams`.

Export your diagram as a pdf (or save one or more screenshots of your diagram) to the `assets-diagrams` folder.

Commit and push your code!

Section 1–3

CodeWars

CodeWars Kata

- **Odd-Even String Sort**
  - Given a string - you must re-order the letters so that "even" letters are at the beginning and "add" letters are at the end
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/580755730b5a77650500010c/java

Module 2

Interface-Based Programming

Section 2–1

Interfaces

Interfaces

- **Interfaces in Java are collections of abstract methods**
  - They are used to define a set of features that a class ***must*** implement if it implements the interface
- **Interfaces do *not* have data members**

Example

IMovable.java

```
public interface IMovable {
   Point move(int xUnits, int yUnits);
   void goHome();
}
```

- **Interfaces can't be instantiated**
- **Instead, interfaces are implemented by classes**

Implementing an Interface

- **A class implements the interface using the `implements` keyword**

Example

```
public class Turtle implements IMovable {
     private String name;
     private Point currentLocation;

      public Turtle(String name) {
         this.name = name;
         this.currentLocation = new Point(25, 25);
         this.power = 100;
     }
      // getters and setters not shown

     public Point move(int xUnits, int yUnits) {

      // the turtle moves the number of units specified in
      // the direction specified

         currentLocation.setX(currentLocation.getX() + xUnits);
         currentLocation.setY(currentLocation.getY() + yUnits);

         return currentLocation;
     }

    public void goHome() {
         this.currentLocation = new Point(25, 25);
      }
}
```

- **If a class says it implements an interface, but the compiler doesn't find an implementation for the interface methods, it generates an error**

- **Each class that implements the interface can choose their own way to implement the interface's behavior**

Example

```
public class Robot implements IMovable {

     private String name;
     private Point currentLocation;
     private int power;

     public Robot(String name) {
        this.name = name;
        this.currentLocation = new Point(0, 0);
        this.power = 100;
     }
     // getters and setters not shown

     public Point move(int xUnits, int yUnits) {

     // the robot can only move the number of units
     // if it has the appropriate power

        int biggestUnit = (xUnits >= yUnits) ? xUnits : yUnits;

        if (power >= biggestUnit) {
              currentLocation.setX(currentLocation.getX() + xUnits);
              currentLocation.setY(currentLocation.getY() + yUnits);

             power -= biggestUnit;
        }

        return currentLocation;
     }

     public void goHome() {
        this.currentLocation = new Point(0, 0);
     }

}
```

Why Interfaces?

- **There are several reasons that you might decide to use interfaces in your application**

- **1 - They can enforce uniformity amongst a set of classes where inheritance doesn't make sense**
  - For example, in Java, the `Collection` interface is implemented by several collection classes

Example

```
// NOTE:  This example does NOT include all of the code in
//        the Collection interface
```

```
public interface Collection<T> extends Iterable<T> {

    int size();
    boolean isEmpty();

    boolean add(T e);
    boolean remove(Object o);
    void clear();

    boolean contains(Object o);
    boolean  containsAll(Collection<?> c);

    Object[] toArray();

    boolean equals(Object o);
    int hashCode();

    // Not all code shown
}
```

- **Classes that implement `Collection` include `ArrayList`, `LinkedList`, and `PriorityQueue`**

- **2 - They allow us to define features implemented by one or more methods and then pick and choose which features make sense for our classes**

Example

`Movable.java`

```
public interface Movable {
    Point move(int xUnits, int yUnits);
    void goHome();
}
```

`Drawable.java`

```
public interface Drawable {
    void draw();
}
```

`Cleaner.java`

```
public interface Cleaner{
    void clean();
}
```

- **Java doesn't support multiple inheritance of classes, but you can implement many interfaces**

Example

RobotVacuum.java

```
public class RobotVacuum implements Movable, Cleaner {
    // code here
}
```

Robot.java

```
public class Robot implements Movable {
    // code here
}
```

EtchASketch.java

```
public class EtchASketch implements Drawable, Cleaner {
    // code here
}
```

Exercise

EXERCISE 1

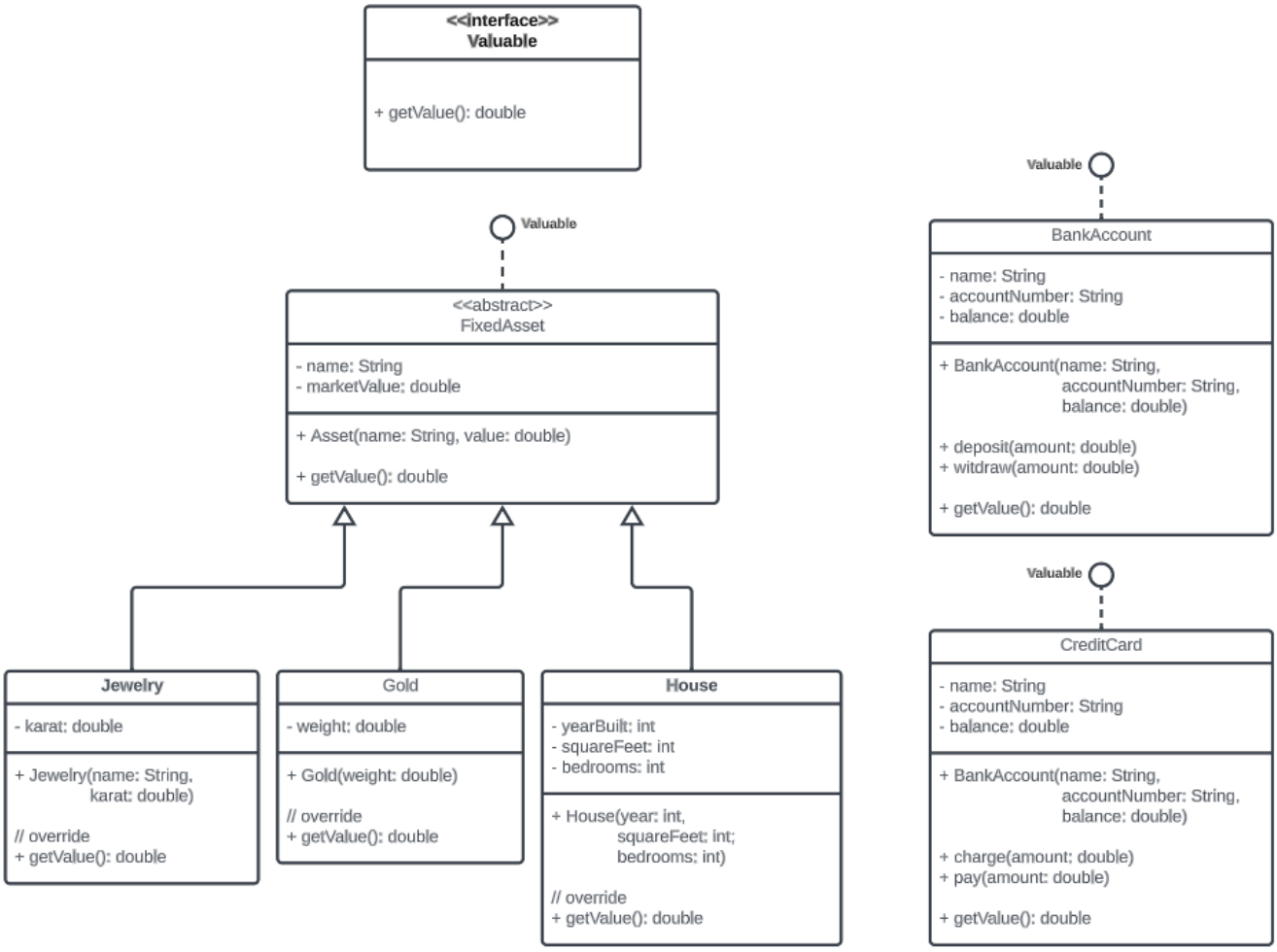
Part 1

Create a new Java application in the `workbook-6` folder named Portfolio.

Add the main application code to the `com.pluralsight` package, but also create a `com.pluralsight.finance` package. You will create all classes related to finance in that package

Create all of the classes that you designed in your Module 1 exercise except for the `Portfolio` class. You will create the `Portfolio` class in a later Exercise.

Your diagram should have included something similar to the following classes.



Part 2

Create a new class named `FinanceApplication` and add the `static void main` method.

Add the following code to the `main` method.

```
BankAccount account1 = new BankAccount(123, "Pam", 12500);
Valuable account2 = new BankAccount(456, "Gary", 1500);

// try to deposit money into both accounts
account1.deposit(100);
account2.deposit(100);
```

Did the deposit work for both accounts? Why or why not?

What methods are available for account1?

What methods are available for account2?

Why?

Commit and push your code!

Section 2–2

Working with Interfaces

Interfaces and Heterogeneous Collections

- **When you have heterogeneous collections, you can query to see of an object implements an interface**
  - If it does, cast the object to the instance type and then use the methods of the interface
  - An ArrayList of type Object can contains ANY type, because all types extend Object

Example

```
ArrayList<Object> things = new ArrayList<Object>();

things.add(new Person());
things.add(new Robot());
things.add(new EtchASketch());
things.add(new Car());
things.add(new RobotVacuum());

for(int i = 0; i < things.size(); i++) {

   if (things.get(i) instanceof Cleaner) {
       Cleaner cleaner = (Cleaner) things[i];
       cleaner.clean();
    }
}
```

Interfaces and Default Methods

- **Java 8 introduced the ability to define default methods in interfaces**
  - A default method defines an implementation that can be overridden in the class if need be
- **This feature was created so that collections could work with Java 8's new support for lambda expressions**
  - Java 8 added a `forEach` method to `List` and `Collection` interfaces
  - Legacy classes that already implemented these interfaces would break with these new additions to the interface
    - Java had to introduce default method implementation of interfaces to support existing code

Example

```
public interface Drawable {
   void draw();

   default void print() {
      System.out.println("This object can draw things.");
   }
}
```

Multiple Defaults

- **There is a possibility that a class is implementing two interfaces with same default methods**

- **This would introduce ambiguity**
  - This is the reason Java did not allow multiple class inheritance

Example

Movable.java

```
public interface Movable {

   void move(int xUnits, int yUnits);
   void goHome();

   default void print() {
      System.out.println("I can move!");
   }
}
```

Cleaner.java

```
public interface Cleaner {

   void clean();

   default void print() {
      System.out.println("I can clean!");
   }
}
```

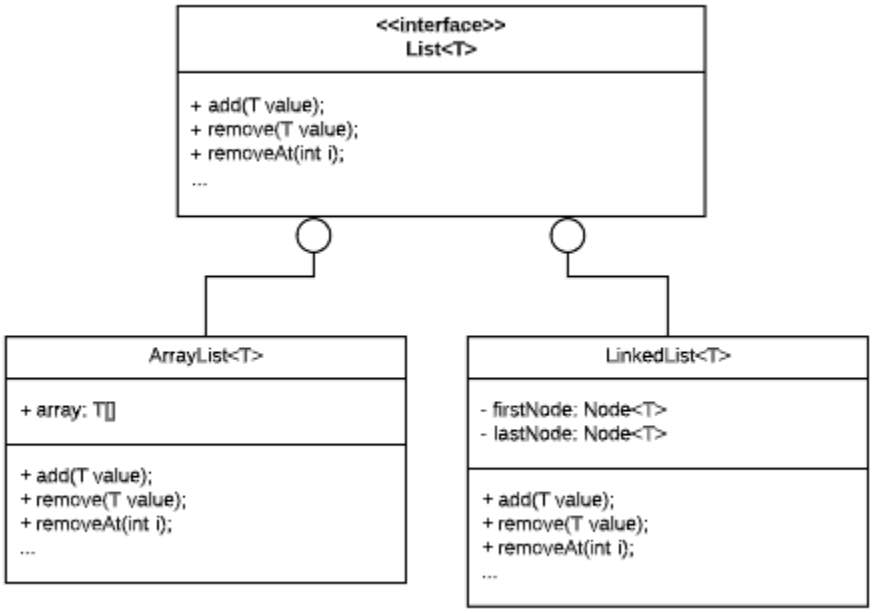
- **To solve this ambiguity, you would need to override the default methods and then call whichever inherited default(s) you wanted**

Example

```
public class RobotVacuum implements Movable, Cleaner {

   public void move(int xUnits, int yUnits) {
     // code not shown
   }

   public void goHome() {
     // code not shown
   }

   public void clean() {
     // code not shown
   }

   public void print() {
      System.out.println("I am a robot vacuum!");
      Movable.super.print();
      Cleaner.super.print();
   }
}
```

Java `List` Interface

- **The `java.util` package contains Java's collection classes and interfaces**



- **Java collections are a convenient way to work with arrays**
  - `ArrayList` uses `Arrays` internally to manage the collection
  - `LinkedList` uses a different structure (non contiguous objects) to manage the collection

- **So far we have been using ArrayList directly when working with collections**
  - Generally when we work with collections we care more about the behavior of the collection than how the data is stored

Java `List` Interface *cont'd*

Example

```
ArrayList<String> names;
names = new ArrayList<>();
names = new LinkedList<>(); // this will cause a compile error
```

NOTE: When we declare a variable as an `ArrayList` you must create a new `ArrayList`.

- **When you declare a collection as a `List` type, you have flexibility to change which kind of list it is.**
  - You are not limited `ArrayList` or even `LinkedList`
  - You can even create a new class that implements the `List` interface

Example

```
List<String> names;
names = new ArrayList<>();
names = new LinkedList<>();
```

- **In Java it is an industry standard to declare collections by using the interface for the collection**

Exercise

EXERCISE 2

Create a new project named WorkingWithInterfaces.

Rather than define an interface in this application, you will implement a Java interface in a custom class.

The interface we are interested in is `Comparable`. It has one method:

```
int compareTo(T o);
```

Thanks to the `compareTo` method of the `Comparable` interface, java can compare one custom class to another. This is necessary if you want to sort a list of objects (such as a `List<Person>`)

You can read the documentation on it here:
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

The `Comparable` interface is implemented by user-defined Java classes when you want to be able to sort collections contain that type of object.

See the following for hints:
https://beginnersbook.com/2017/08/comparable-interface-in-java-with-example/

In this exercise, you will build a Java application that defines a class named `Person`.

A `Person` object has a `firstName`, a `lastName`, and an `age`.

In the `main()`, you will build an `ArrayList` of `Person` objects similar to that below:

```
List<Person> myFamily - new ArrayList<Person>();
myFamily.add( new Person("Dana", "Wyatt", 63) );
myFamily.add( new Person("Zachary", "Westly", 31) );
myFamily.add( new Person("Elisha", "Aslan", 14) );
myFamily.add( new Person("Ian", "Auston", 16) );
myFamily.add( new Person("Zephaniah", "Hughes", 9) );
myFamily.add( new Person("Ezra", "Aiden", 17) );
```

You want to sort the list by last name and display the results. Try to use the built in `sort` method.

```
Collections.sort(myFamily)
```

Did it work?

How would Java know how to compare one person to another, or that the `lastName` property should be used for sorting? It doesn't.

In order to sort, your `Person` class needs to implement the `Comparable` interface and add code to the `compareTo` method.

Write code to implement the `Comparable` interface then try to sort and display your list of people again.

BONUS:

If two people have the same last name, sort by first name.

What if there are two people with the same first and last name? Could you also sort by Age?

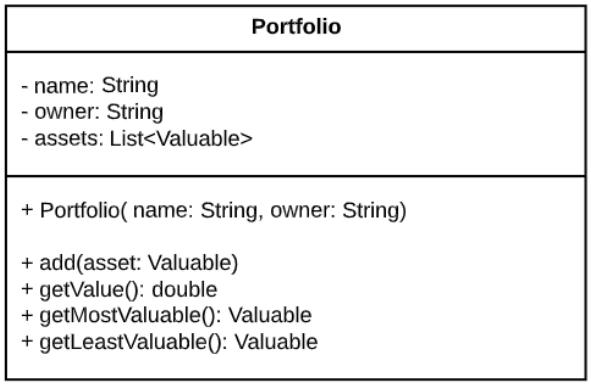
Commit and push your code!

EXERCISE 3

Continue working with the Portfolio project from exercise 1.

Part 1

Add a portfolio class as noted below:



The `getValue()` method should return the net value of all assets.

The `getMostValuable()` method should return the asset that has the highest value.

The `getLeastValuable()` method should return the asset that has the lowest value, possibly a credit card with the highest balance.

Bonus

Modify the main method of your application to prompt users to add new assets to their portfolio. The application should promt the user for the type of asset and all of it's values

Commit and push your code!

Section 2–3

CodeWars

CodeWars Kata

- **Decimal Time Conversion**
  - Convert the time (hours and minutes - h:mm) into their decimal value
    - 2:30 -> 2.5
    - 3:15 -> 3.25
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/6397b0d461067e0030d1315e/java

Module 3

Generics

Section 3–1

Generics

Generics

- **Generics were added to Java in JDK 5.0**
  - They are a major part of many popular programming languages, including C++, C#, TypeScript and Swift
- **Generic can be applied to classes or methods**
- **When applied to a class, generics allow you to generalized the data type of member variable(s)**
  - They type of the variable(s) are usually specified using a letter like **`T`** and then replaced with an actual data type like `Integer`, `String` or a user-defined data type when the `object` is instantiated
- **When applied to a method, generics allow you to generalized the data type of parameters and return types**
- **IMPORTANT NOTE: Java *does not* support using *primitive types* with generics**
  - This means you must use one of Java's wrapper's for primitive types
    * `Integer` for `int`
    * `Double` for `double`
    * `Character` for `char`
    * `Boolean` for `bool`
    * etc

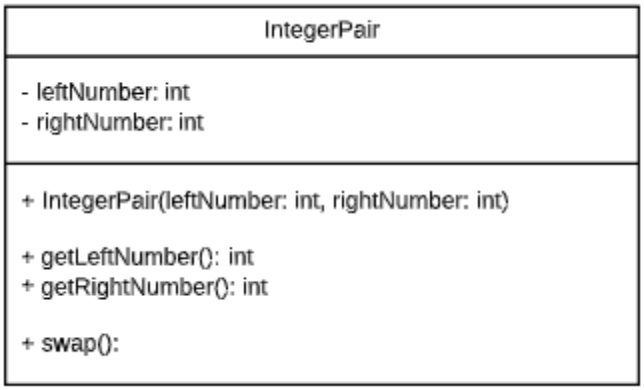
The need for Generics

- **Inheritance gives us the ability to generalize properties and methods into a parent class**
  - It removes duplication and makes our code more re-usable

- **Inheritance by itself works well when the internal data types are well known and unique**
  - But what if we need the same functionality, but with different data types

Example

This `IntegerPair` class has functionality to work with pairs of whole numbers



Example

```
public class IntegerPair {

  // A pair contains two integers
  private int leftNumber;
  private int rightNumber;

  public IntegerPair(int leftNumber, int rightNumber) {
    this. leftNumber = leftNumber;
    this. rightNumber = rightNumber;
  }

  public int getLeftNumber ()  {
    return this. leftNumber;
  }

  public int getRightNumber ()  {
    return this. rightNumber;
  }

  public void swap() {
    int temp = leftNumber;
    leftNumber = rightNumber;
    rightNumber = temp;
  }

}
```

- **What happens when we need pairs of other objects?**
  - Pairs of doubles, Strings, LocalDates
  - What about custom classes: Person, Employee, Vehicle
  - We would need to create a new class for EACH data type for which we want to manage the pair logic


IntegerPair
- leftNumber: int
- rightNumber: int
+ IntegerPair(leftNumber: int, rightNumber: int)
+ getLeftNumber(): int
+ getRightNumber(): int
+ swap():
DoublePair
- leftNumber: double
- rightNumber: double
+ DoublePair(leftNumber: double, rightNumber: double)
+ getLeftNumber(): double
+ getRightNumber(): double
+ swap():
LocalDatePair
- leftDate: : LocalDate
- rightDate: LocalDate
+ LocalDatePair(leftDate: LocalDate, rightDate: LocalDate)
+ getLeftDate(): LocalDate
+ getRightDate(): LocalDate
+ swap():
StringPair
- leftString: String
- rightString: String
+ StringPair(leftString: String, rightString: String)
+ getLeftString(): String
+ getRightString(): String
+ swap():
EmployeePair
- leftEmployee: Employee
- rightEmployee: Employee
+ EmployeePair(leftEmployee: Employee,
rightEmployee: Employee)
+ getEmployeeString(): Employee
+ getEmployeeString(): Employee
+ swap():
VehiclePair
- leftVehicle: Vehicle
- rightVehicle: Vehicle
+ VehiclePair(leftVehicle: Vehicle,
rightVehicle: Vehicle)
+ getLeftVehicle(): Vehicle
+ getRightVehicle(): Vehicle
+ swap():

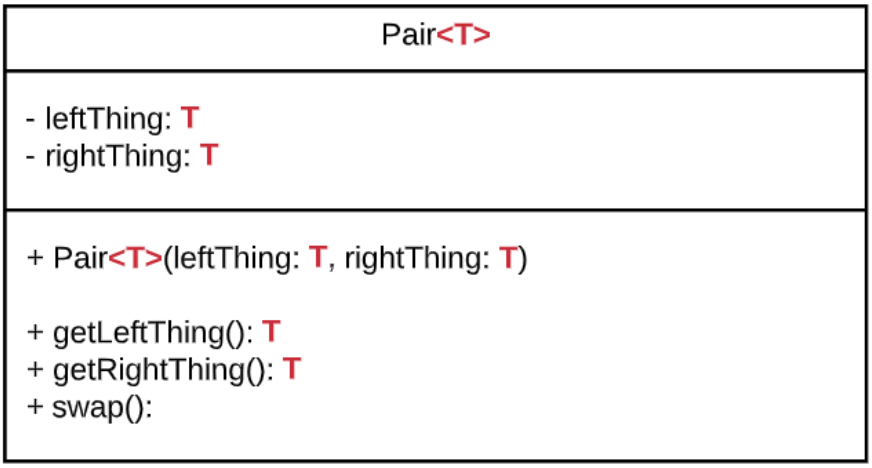
Section 3–2

Introducing Generics

What are Generics

- **A generic is a way to define re-usable functionality without a pre-defined data type**

- **Write functionality that is *generic* to all data types, but performs a specific task**
  - Since we don't know the data type, we replace it with `<T>`
    - **T** just refers to type, but can be any letter
    - Notice that **T** is used inside the class anywhere we need to refer to the data type



Generic Classes

- **Creating a generic class is a matter of:**
  - Placing the generic parameter(s) in < > after the name of the class
  - Using the generic parameter(s) everywhere you expect a specific parameter once an object is instantiated

Example

Pair.java

```
class Pair<T>  {

  // A pair contains two objects
  private T leftThing;
  private T rightThing;

  Pair(T leftThing, T rightThing) {
    this.leftThing = leftThing;
    this.rightThing = rightThing;
  }

  public T getLeftThing()  {
    return this.leftThing;
  }

  public T getRightThing()  {
    return this.rightThing;
  }

  public void swap()  {
    T temp = leftThing;
    rightThing = leftThing;
    leftThing = rightThing;
  }
}
```

- **You can then instantiate objects by proving the generic parameter in < > after the name of the class**

Example

```
Pair<Integer> twoNums = new Pair<Integer>(63, 65);
System.out.println(twoNums.getLeftThing()) // 63
twoNums.swap();
System.out.println(twoNums.getLeftThing()) // 65

Pair<String> twoWords = new Pair<String>("Me", "You");
System.out.println(twoWords.getLeftThing()) // "Me"
twoWords.swap();
System.out.println(twoWords.getLeftThing()) // "You"
```

Example

```
public class MainApp
{
    public static void main (String[] args) {

        // Create an instance of a Pair<T> where T is a String
        Pair<Integer> twoNums = new Pair<Integer>(63, 65);
        System.out.println(twoNums.getLeftThing() +
           " - " + twoNums.getRightThing());

        // Create an instance of a Pair<T> where T is an
        Pair<String> twoWords = new Pair<String>("Me", "You");
        System.out.println(twoWords.getLeftThing() +
           " n " + twoWords.getRightThing());
    }
}
```

OUTPUT

```
63 - 65
Me n You
```

Generic Methods

- **Generic methods use generic parameters and return types**
- **You can use generic methods in generic classes, as you saw in the previous examples**
- **You can also use generic methods in standard Java classes**
  - This is useful if the entire class is not dependent on the generic type
  - If you use the generic parameter as a return type, you must surround it with < >

Example

Labeler.java

```
public class Labeler  {
   // A generic method that displays a label and a value
   static <T> void displayWithLabel(String label, T
value)  {
      System.out.println(label +  ": " + value);
   }
}
```

MainApp.java

```
public class MainApp {
   public static void main (String[] args) {
      Labeler labeler = new Labeler();

      // Calling generic method with String argument
      labeler.displayWithLabel("Name", "Dana");

      // Calling generic method with Integer argument
      labeler.displayWithLabel("Age", 63);
   }
}
```

OUTPUT

```
Name: Dana
Age: 32
```

Section 3–3

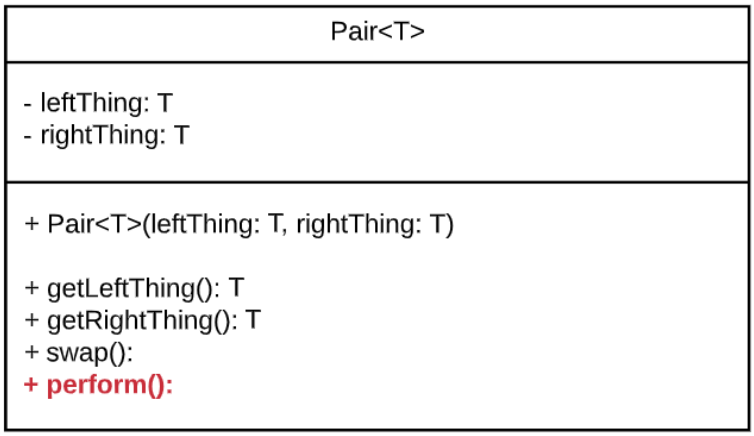
Limiting Generic Types

Generic Limitations

- **The purpose of a generic class is to create common functionality**

- **The `Pair` class did not do anything unique with any of the data types**
  - The only thing it did was to swap the left and right items

- **Sometimes you want to execute specific methods of the left and right things**
  - What if you had a Pair<Musician> and you want them to perform



- **We need to add a `perform` method**
  - But we cannot guarantee that our variables have the perform method

Example

`Pair.java`

```
class Pair<T>  {
  // A pair contains two objects
  private T leftThing;
  private T rightThing;

  Pair(T leftThing, T rightThing) {
    this.leftThing = leftThing;
    this.rightThing = rightThing;
  }

  public T getLeftThing()  {
    return this.leftThing;
  }

  public T getRightThing()  {
    return this.rightThing;
  }

  public void swap()  {
    T temp = leftThing;
    rightThing = leftThing;
    leftThing = rightThing;
  }

  public void perform() {

    // we cannot guarantee that the left and right things
    // have a perform  method defined
    // these lines of code would not compile
    leftThing.perform();  //error
    rightThing.perform(); //error
  }

}
```

- **This logic is reasonable if we created a `Pair<Musician>`**
  - This assumes that the Musician interface defines a `perform` method.

Example

Musician.java

```
public class Musician {
  void perform();
}
```

Program.java

```
Pair<Musician> duet = new Pair<Musician>(new Violinist(),
                                         new Cellist());
duet.perform();
```

- **But what if we create other types of pairs?**

Example

Program.java

```
Pair<Integer> numbers = new Pair<Integer>(15, 32);

// Integers do not KNOW HOW to perform
//   - this would throw an exception
numbers.perform();
```

Making Generics more Specific

- **We can force Generic classes to recognize methods**
  - We can specify which classes or interfaces our Generics will allow
  - This is done by limiting the definition of T
    - `<T extends Musician>`
    - Now a Duet can only be created for classes which extend Musician

Example

Duet.java

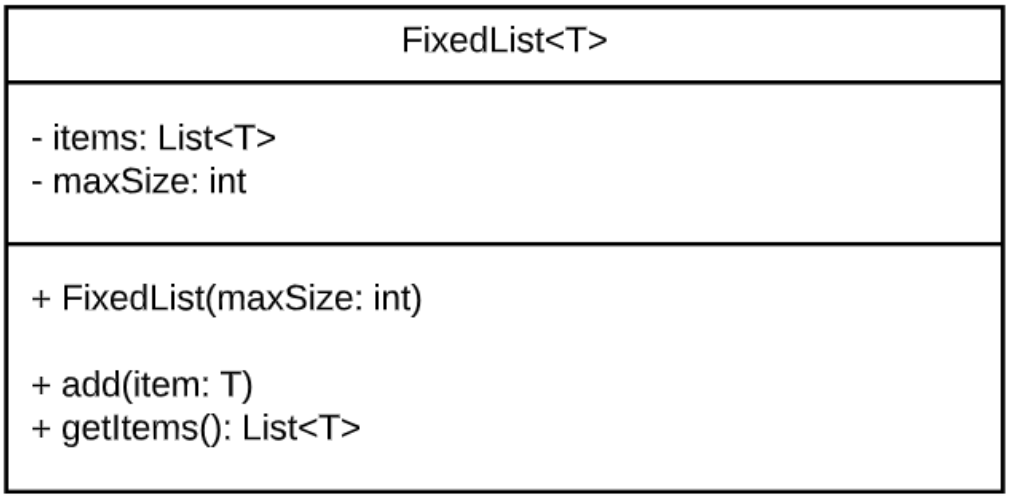
```
class Duet<T extends Musician>  {

  private T left;
  private T right;

  Duet(T left, T right) {
    this.left = left;
    this.right = right;
  }

  public T getLeft()  {
    return this.left;
  }

  public T getRight()  {
    return this.right;
  }

  public void perform() {

    // because T Is-A Musician we know that they
    // will have the perform method
    left.perform();
    right.perform();
  }
}
```

Exercises

EXERCISE 1

Create a new Java application in the `workbook-6` folder named CustomCollections. Add a `com.pluralsight.collection` package.

In your project you will create a new class called `FixedList<T>`. The list will allow you to create a collection of any type of item. But unlike a regular `ArrayList`, you will limit the maximum number of items that can be added to your list.



The constructor should take a single argument which specifies the maximum number of items.

Modify the `add` method to ensure that your list cannot grow bigger than the `maxSize` that was specified by the constructor.

In your `main` method, write code to test creating different types of lists. Also test the logic of your add method.

```
FixedList<Integer> numbers = new FixedList<>(3);
numbers.add(10);
numbers.add(3);
numbers.add(92);
numbers.add(43); // this line should fail

System.out.println(numbers.getItems().size());

FixedList<LocalDate> dates = new FixedList<>(2);
dates.add(LocalDate.now());
dates.add(LocalDate.now());
dates.add(15); // this line should fail
```

Section 3–4

CodeWars

CodeWars Kata

- **Speed Limit**
  - Calculate the price of a speeding ticket based on the speed of the car
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/635a7827bafe03708e3e1db6/java

Module 4

Java Streams

Section 4–1

Collections vs Streams

Collections and Streams

- **Collections are an integral part of any Java application**
- **A collection is often an Array or List of some sort**
  - List of Customers
  - List of Orders
  - List of Inventory
- **Collections are also sets of data that need to be processed**
  - Filter to get a sub-set of the list
  - Aggregate the data
    - Calculate the total sales
    - Find the most/least expensive product
    - Count all customers
    - Get the average sales in a specific month
  - Change the shape of the data in the list
    - Input a `List<Order>` and output a `List<PickList>`

Processing Collections with loops

Example

LineItem.java

```
public class LineItem {
  private int productId;
  private String productName;
  private double price;
  private int quantity;

  ... constructor getters and setters

  public double getLineTotal(){
    return quantity * price;
  }
}
```

Order.java

```
public class Order {
  private int id;
  private String customer;
  private List<LineItem> lineItems;

  public Order(int id, String customer){
    this.id = id;
    this.customer = customer;
    this.lineItems = new ArrayList<>();
  }

  public void addItem(LineItem item){
    lineItems.add(item);
  }

  public double getTotal(){
    double total = 0;
    for(LineItem item: lineItems) {
      total = item.getLineTotal();
    }
    return total;
  }
}
```

Example

Get all orders under $25.

```
List<Order> allOrders = new ArrayList<>();
// populate all ordrders

List<Order> impulseOrders = new ArrayList<>();

// loop through all orders
// and add them to the list of impulse buys
for(Order order: allOrders){

  if(order.getTotal() < 25){
    impulseOrders.add(order);
  }
}
```

Example

Calculate the total sales for all orders.

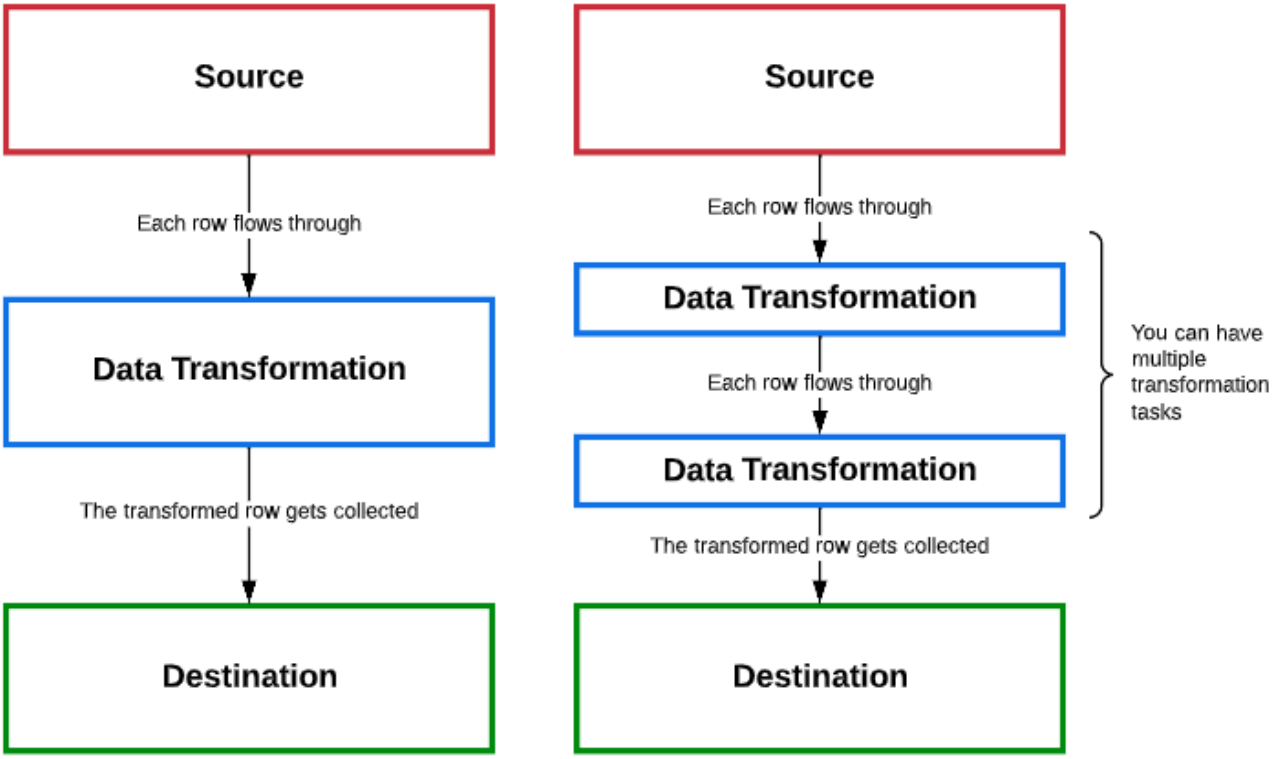
```
List<Order> allOrders = new ArrayList<>();
// populate all ordrders

double salesTotal = 0;

// loop through all orders
for(Order order: allOrders){
  salesTotal += order.getTotal();
}

System.out.println("Total: $" + salesTotal);
```

Introduction to Streams

- **Java 8 introduced Streams to simplify processing collections**

- **We manage data in 2 basic states**
  - Data at rest
    - This is data that is not currently in use
    - It refers to data in a database, saved in a file, but also data in memory that is stored in a collection
  - Data in transit
    - This is data that is currently being processed (usually as a stream)
    - Think about the file streams that are used to read a text or csv file when the data is being loaded into memory

- **Data that is being processed goes through 3 stages**
  - Data at rest -> must be funneled into a stream
    - This is the source
  - Data in transit -> flows through the stream and any tasks in the stream
    - These are the tasks the manipulate the data
    - There can be more than one transformation task
  - Data in transit -> must be collected back into data at rest
    - This is the destination
    - This could be another collection or a single variable


Source
Each row flows through
Data Transformation
The transformed row gets collected
Destination
Source
Each row flows through
Data Transformation
Each row flows through
Data Transformation
You can have multiple transformation tasks
The transformed row gets collected
Destination

Exercises

In this exercise you will create an application that allows you to search for people by name. In exercise 1 you will do it without the use of Java streams, instead you will program the code using traditional for loops. You will convert to using java streams in a later exercise.

EXERCISE 1

Create a new Java application in the `Mod03` folder named Streams. Add a `com.pluralsight.streams` package.

Create a `Person` class with the following properties: `firstName, lastName, age`.

Create a `Program.java` file and add your `static void main` method.

Step 1

In your main method create a list of at least 10 people.

You can use ChatGPT or an online random name generator to create a list (https://randomwordgenerator.com/name.php)

Step 2

Prompt the user for a name to search (first or last).

Using a for loop, create a new `List` of people whose name was a match, display the names of the people in that list

Step 3

Calculate the average age of all people in the original list and display it.

Display the age of the oldest person in the list.

Display the age of the youngest person in the list.

Section 4–2

Lambda Expressions

Lambda Expressions

- **One of the most anticipated features added in Java 8 was lambda expressions**
  - These are dynamic **inline functions** (or anonymous functions)
- **A lambda expression is a function in disguise**
  - It is passed as a parameter to another function

Syntax

```
parameter -> expression body
```

or

```
(parameter1, parameter2, …) -> expression body
```

- **Rules for lambda expressions include:**
  - Optional parameter type declaration − the compiler infers the type from the value of the parameter
  - Parenthesis are only needed around parameters if there is more than one parameter
  - Curly braces are only needed around the expression body if the body contains more than one statement
  - No return statement needed if the expression body is just a single expression

Using Lambdas with Collections

- **Java 8 added a `stream()` method to the `Collection` interface that returns the collection in a way that will work with lambdas**

Example

```
ArrayList<Employee> employees = new ArrayList<>();
...

Stream<Employee> = employees.stream();
```

- **The Stream provides a pipeline that you can use to provide processing on the elements in the stream**
  - https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html
  - In the following example, we will examine `filter()`

Example

```
import java.util.*;
import java.util.stream.Collectors;

// Use the List interface as the type of employees
// so that we can work with some of the generic methods
List<Employee> employees = new ArrayList<>();
...

List<Employee> matchingEmps = employees.stream()
       .filter(p -> p.getLastName().equals(lastName))
       .collect(Collectors.toList());

System.out.println("Matching: " + matchingEmps);
```

Lambdas and `forEach()`

- **Java 8 introduced `forEach()` for collections**
  - The lambda expression is executed for ***each*** element in the collection

Example

```
import java.util.ArrayList;

public class Program {

   public static void main(String[] args) {

      List<String> names = new ArrayList<String>();
      names.add("Ezra");
      names.add("Ian");
      names.add("Siddalee");
      names.add("Elisha");
      names.add("Pursalane");
      names.add("Zephaniah");

      names.forEach(name -> {
          System.out.println(name);
      });
   }
}
```

- **The lambda expression that is passed to `forEach` could be multiple lines long**

Example

Employee.java

```
public class Employee {
   private String name;
   private String jobTitle;
   private double salary;

   public Employee(String name, String jobTitle,
   double salary) {
      this.name = name;
      this.jobTitle = jobTitle;
      this.salary = salary;
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double salary) {
      this.salary = salary;
   }
}
```

Program.java

```
import java.util.ArrayList;

public class Program {

   public static void main(String[] args) {

      List<Employee> emps = new ArrayList<Employee>();
      emps.add(new Employee("Ezra", "Actor", 72750.0));
      emps.add(new Employee("Ian", "Banker", 252750.0));
      emps.add(new Employee("Siddalee", "Model", 1500000.0));
      emps.add(new Employee("Elisha", "Programmer", 103500.0));
      emps.add(new Employee("Pursalane", "Teacher", 697250.0));
      emps.add(new Employee("Zephaniah", "Engineer", 136000.0));

      emps.forEach(emp -> {
         if (emp.getSalary() < 100000) {
            emp.setSalary(101000.0);
         }
         else {
            emp.setSalary(emp.getSalary() * 1.1);
         }
         System.out.println(emp.getName() + " earns $" +
                            String.format("%.2f", emp.getSalary());
      });
   }
}
```

Section 4–3

Java Stream Methods

Generating Streams and Collecting Results

- **With Java 8, the `Collection` interface has two methods to generate a `Stream`**
  - `stream()` returns a sequential stream of the collection data
  - `parallelStream()` returns a parallel stream of the collection data used in asynchronous programming
    - Parallel streams are beyond the scope of this class

Example

```
List<String> states = Arrays.asList(
      "Alabama", "Alaska", "Arizona", "Arkansas",
      "California", "Colorado", "Connecticut",
      /* others not shown */ );

String letter = "C";

List<String> matchingStates = states.stream()
             .filter(state -> state.startsWith(letter))
             .collect(Collectors.toList());
```

- **Collectors combine the result of processing on the elements of a stream**
  - `Collectors.toList()` returns a `List<T>` where `T` is the type of data collected
  - If you want to collect the results into an `ArrayList`, you can use a different collector
    - In this case, use `Collectors.toCollection(ArrayList::new)`

Example

```
public ArrayList<String> getMatchingStates(String firstChars) {

   // this.states is a class-level ArrayList<String>
   ArrayList<String> matchingStates = this.states.stream()
             .filter(state -> state.startsWith(firstChars))
             .collect(Collectors.toCollection(ArrayList::new));

   return matchingStates;
}
```

filter() and count()

- **The `filter()` method reduces the stream to only the elements that match a condition**

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

List<String> matching = titles.stream()
     .filter(title -> title.toLowerCase().contains("halloween))
     .collect(Collectors.toList());
```

- **The `count()` method returns a count of the number of items in the stream**
  - In this case, it only returns the count

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

int count = titles.stream()
    .filter(title -> title.toLowerCase().contains("Halloween"))
    .count();
```

forEach()

- **The `forEach()` method calls a named method for each value in the stream**

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

titles.stream()
    .filter(title -> title.toLowerCase().contains("halloween"))
    .forEach(System.out::println);
```

sorted()

- **The `sorted()` method sorts the stream**
  - The `sorted` method can be used with or without a lambda expression
  - The `sorted` method only works with Types that implement the `Comparable` interface
  - https://howtodoinjava.com/java8/stream-sorted-method

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

titles.stream()
   .filter(title -> title.toLowerCase().contains("halloween"))
   .sorted()
   .forEach(System.out::println);
```

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

List<String> sortedMovies = titles.stream()
   .filter(title -> title.toLowerCase().contains("halloween))
   .sorted()
   .collect(Collectors.toList());
```

map()

- **The `map()` method maps each element in the stream to a new result**

Example

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

// get list of unique squares
List<Integer> squaresList = numbers.stream()
             .map(num -> num * num)
             .distinct()
             .collect(Collectors.toList());

System.out.println(squaresList);
```

OUTPUT

```
[9, 4, 49, 25]
```

reduce()

- **The `reduce()` method aggregates all elements into a single value of the same type**
  - Think of aggregate functions like sum, count, min, max etc.
  - The reduce function also acts as the collector.
  - It takes 2 arguments
    * The initial value of the aggregate
    * The lambda expression of how to perform the aggregation

Example

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

// get the sum of all numbers
List<Integer> sum = numbers.stream()
             .reduce(
                      // this is the initial value of temp
                      0,
                      // temp holds the aggregation as all
                      // numbers are added
                      (temp, num)-> temp += num)
                     );

System.out.println(sum);
```

OUTPUT

```
25
```

Exercises

EXERCISE 2

Copy the Streams project and rename the copy to StreamsPart2.

Step 1

In your name search logic, replace your for loop with a `stream` and `filter` function.

Ensure that your new list of names still contains the correct results.

Step 2

Replace the for loop in your average age calculation with stream methods.

HINT: You will need to chain multiple methods together: i.e. `map()`, `reduce()`

Step 3:

Using only stream methods find the following answers.

HINT: This could be done a few different ways: using `sorted()` or a combination of `map()` and `reduce()`

Display the age of the oldest person in the list.

Display the age of the youngest person in the list.

Section 4–4

CodeWars

CodeWars Kata

- **Multiples of 3 or 5**
  - Sum all of the numbers that are multiples of 3 or 5 within a given range
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/514b92a657cdc65150000006/java

Module 5

Java Packages

Section 5–1

Java Packages

Java Packages

- **In Java a package is a folder used to organize your classes/objects**
- **A Java package is a collection of related `.class` files**
  - It organizes the class files in a series of folders
- **Packages are used to avoid naming conflicts with classes**
  - They also encourage you to create more maintainable code by organizing related classes together
- **There are two types of packages:**
  - The Java API packages
  - User-defined packages
- **The Java API consist of a large class library that is organized into packages**
  - For example, we've used classes from the `java.util` and `java.io` packages
  - `https://docs.oracle.com/javase/8/docs/api/`
- **Developers can also create packages of the custom code**
  - A package name must match the folder structure of where the `.java` file is saved

Using `import`

- **You can use a class defined in a different package by using the `import` keyword**
  - You can import a single class

Example

```
import java.util.Scanner;
```

  - You can import all classes from a specific package

Example

```
import java.util.*;
```

User-Defined Packages

- **You can create packages for your own classes**
  - This allows you to group related classes together in a way that helps organize your code

- **Package names are typically expressed in a reverse DNS format**

Example

```
com.pluralsight
```

- **The package name above implies the following folder structure in you Maven project**
  - I.e. you are required to have this folder structure in order to use that package name

Example

```
project-folder
   |
   |-- src
     |
     |-- main
       |
       |-- java
         |
         |-- com
           |
           |-- pluralsight
             |
             |-- java files go here
```

User-Defined Packages *cont'd*

- **In a Maven project your folder structure is pre-defined**
  - Your java code files must be nested inside the `src/main/java` folder
  - You can still create new packages, but they must also be created inside that same folder
- **When the project is compiled, the packages and class files are saved to a different folder**
  - The Maven process creates an output directory named `target/classes`
  - The package folder structure is re-created in that folder
  - All compiled classes are saved in their respective package

User-Defined Packages *cont'd*

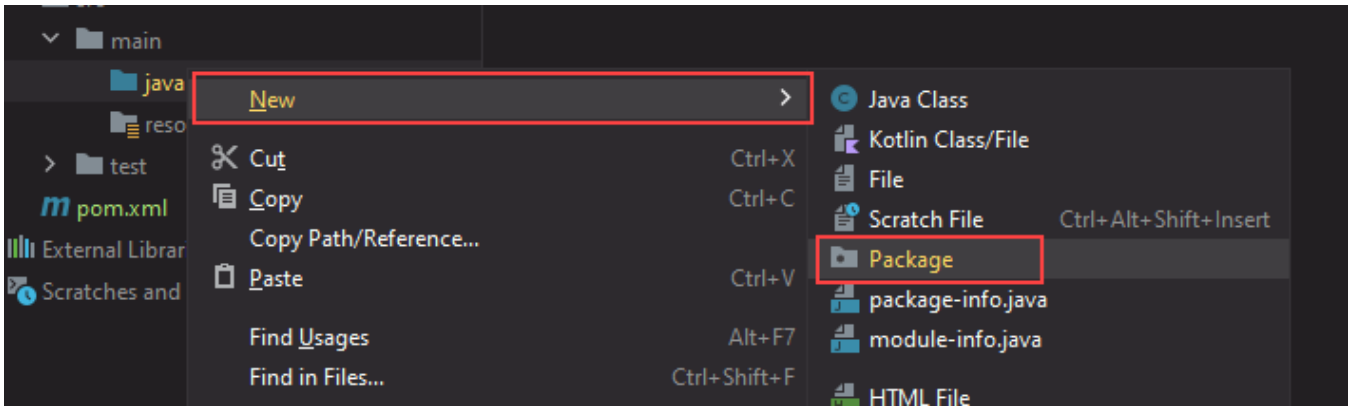
Example

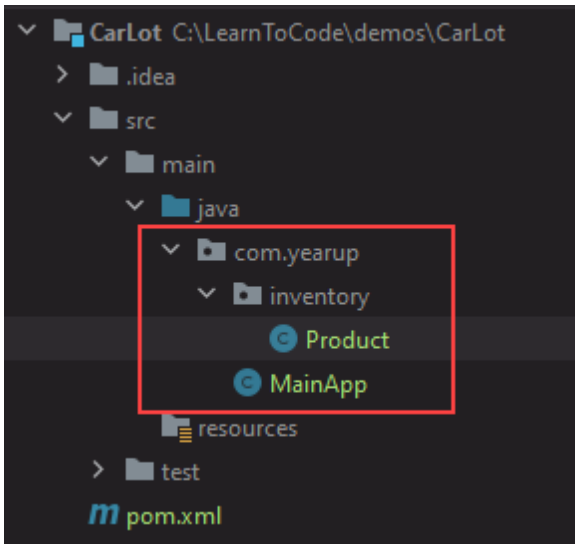
```
project-folder
    |
    |-- src
        |
        |-- main
            |
            |-- java
                |
                |-- com
                    |
                    |-- pluralsight
                        |
                        |-- java files go here
    |
    |-- target
        |
        |-- classes
            |
            |-- com
                |
                |-- pluralsight
                    |
                    |-- class files get compiled here
```

Example: Defining Packages

- First you must create the package folder structure and add the new `Product.java` file in that folder

Example





Example: Defining Packages

- **To specify the name of the package, place a `package` statement as the first statement in the code file**

Example

Product.java

```
package com.pluralsight.inventory;

public class Product {
   private int id;
   private String productName;
   private int qtyOnhand;
   private double price;

   // remaining code not shown
}
```

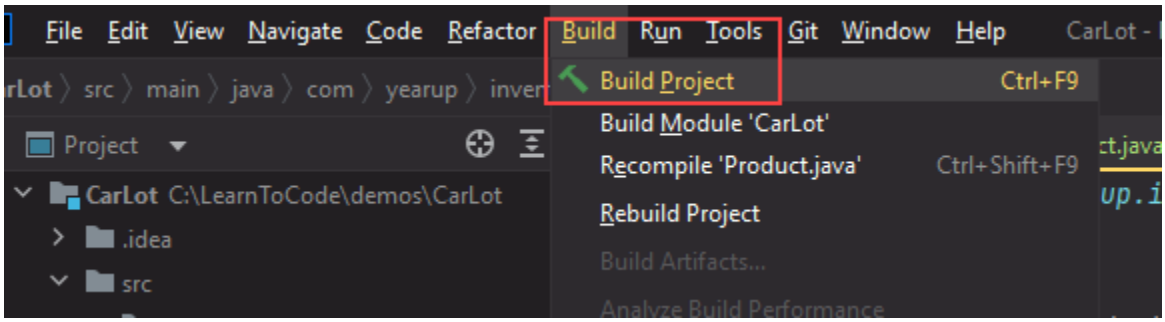
MainApp.java

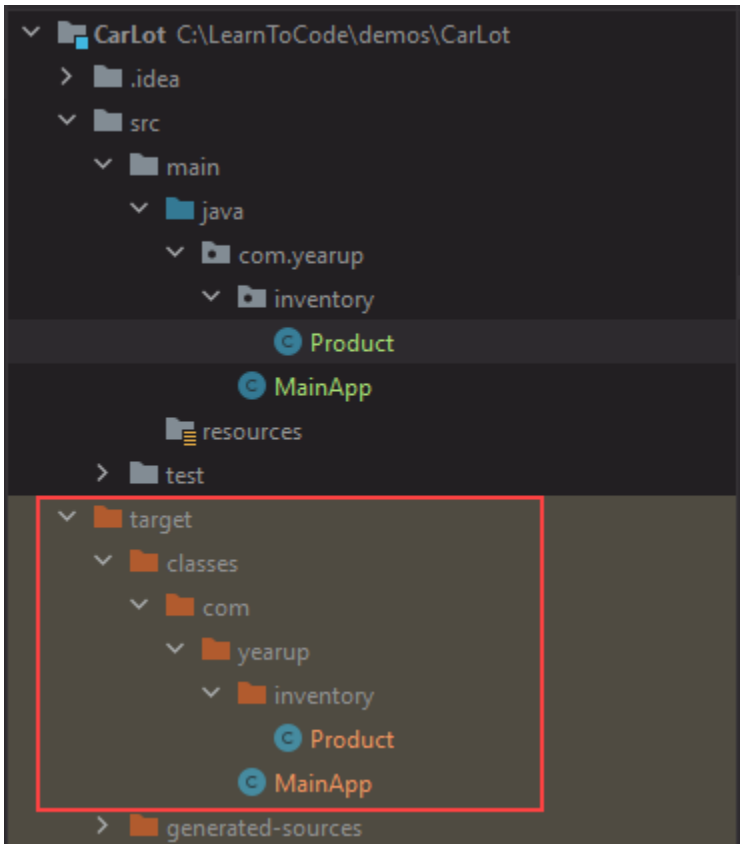
```
package com.pluralsight;

import java.util.ArrayList;
import com.pluralsight.inventory.Product;

public class MainApp {

   public static void main(String[] args) {

      List<Product> inventory = new ArrayList<Product>();
      ...

      // remaining code not shown

   }
}
```

NOTE: Since `MainApp` and `Product` classes are located in different packages, you must import the `com.pluralsight.inventory` package

Compiling Packages

- **To compile the classes and create the package structure go to `Build -> Build Project`**
  - This will compile all java files in the project and create the appropriate output file structure





Launching the Program

- **To run the program outside of IntelliJ, remember to use the full name of the class containing the main method**

Example

From the project folder, issue the following commands

```
cd target/classes
java com.pluralsight.MainApp
```

Exercise

EXERCISE 1

In this exercise, you will modify an existing application and organize it into packages. The project is called TurtleLogo. Find the `TurtlePaint.zip` file in your starter files and unzip it to your `workbook-6` directory. Open the project in IntelliJ.

The TurtleLogo project is a Java adaptation of a programming language that was created in 1967 called Logo. The purpose of the language was to help new programmers learn how to program. These classes were adapted from the implementation at https://codehs.com/sandbox/apcsa/java-turtle.

It is a simple drawing program that works similar to the concept of Etch-A-Sketch. A turtle is your pen and it is up to you to move it around the canvas to draw your logo.

There are three classes in the application.

World

The world is your canvas… literally. It defines the drawing surface. Point `0,0` is the middle of the canvas

Turtle

The turtle is your pen. You can rotate the turtle to the right or to the left. And then you can move forward in the direction you are facing. When you create a new turtle, you specify the world that the turtle will live in, and where it can draw. Here are some of the turtle functions:

```
turtle.penUp();
turtle.penDown();
turtle.turnRight(90); // rotates 90 degrees to the right
turtle.forward(100); // moves 100 "steps" forward
```

MainApp

This is the location of the static void main where you will write your instructions to create your logo.

When you open your project in IntelliJ you will notice that all 3 java files are nested directly in the `java` folder.

Java classes should not directly be in the `java` folder but should be organized into packages. Your first task is to create the following package structure and place the classes in their appropriate package.

NOTE: As you create the packages and move the java classes into their appropriate package, IntelliJ will refactor the code for you and add the ne

```
TurtlePaint
   |
   |-- src/
     |
     |-- main/
       |
       |-- java/
         |
         |-- com/
           |
           |-- pluralsight/
             |
             |-- forms/
               |
               |-- World.java
               |
               |-- Turtle.java
             |
             |-- MainApp.java
```

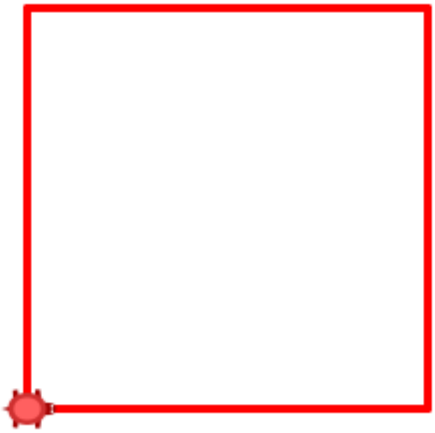
EXERCISE 2

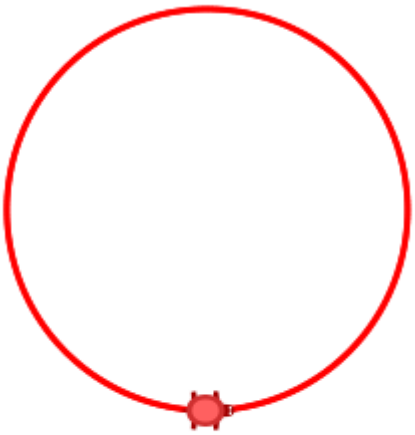
Look through the functions that are available in the Turtle class, and experiment with what your functions can do.

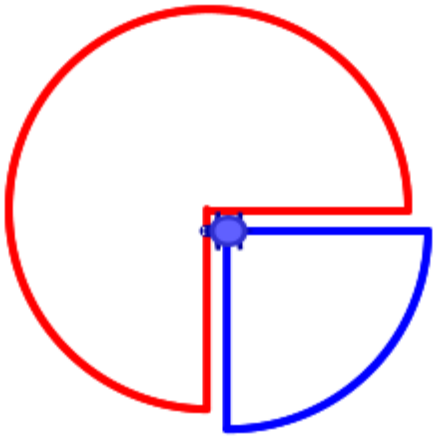
For example, the `turtle.drop()` function allows you to specify a path to an image and it will be placed on the world canvas.

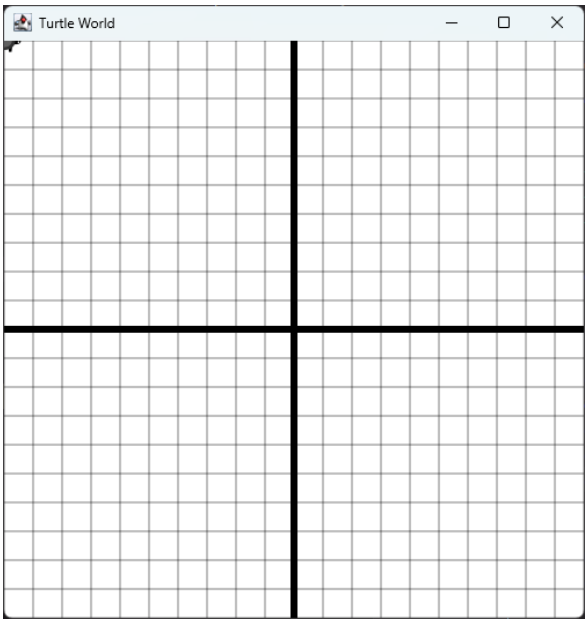
The World class also has some interesting functions, such as `world.save()` which will allow you to save a jpg or png of your logo.

Try to draw some of the following shapes:









Can you draw a Tic-Tac-Toe game?

Can you create a block letter of your first initial?

Or write your name?

Commit and push your code!

Section 5–2

CodeWars

CodeWars Kata

- **Duck, Duck, Goose**
  - Given an array of players, you need to return the name of the selected player
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/582e0e592029ea10530009ce/java