CS1020 Lecture Note #3: **Object Oriented Programming Part 2**

More OOP concepts

Lecture Note #3: **OOP Part 2**

- **Objectives:**
  - Introduce Inheritance and Polymorphism and related concepts
  - Introduce generics, allow operations that are not tied to a specific data type
  - Introduce vector class
- **References:**
  - Wrapper classes:
    - Chapter 1, Section 1.1, pages 29 to 30
  - **Object** class:
    - Chapter 1, Section 1.5, pages 56 to 58
  - **Generics:**
    - Chapter 9, Section 9.4, pages 499 to 507

1.1 Class Inheritance

- Inheritance: To derive new classes by extending existing classes
- When a class c1 is derived from another class c2, then

  c1 is called a subclass (child class) of c2, and

  c2 is called the super-class (parent class) of c1.

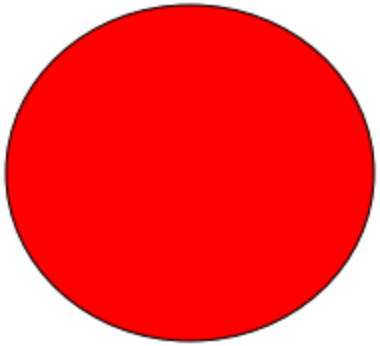
1.1 Class Inheritance (in UML notation)

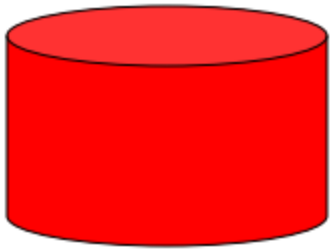
Superclass

Circle
- radius: double
+ getRadius (): double
+ setRadius (radius: double): void
+ findArea(): double

Subclass

Cylinder
- length: double
+ getLength(): double
+ setLength(length: double): void
+ findVolume(): double





class CircleWithAccessors

```
// CircleWithAccessors.java: The circle class with accessor methods
public class CircleWithAccessors {
   private double radius;
   public CircleWithAccessors()              // default constructor
      { this (1.0); }
   public CircleWithAccessors(double r)      // constructor
      { radius = r; }
   public double getRadius()                 // accessor
      { return radius; }
   public void setRadius(double newRadius) // mutator
      { radius = newRadius;}
   public double findArea()                  // facilitator
      { return radius * radius * 3.14159;}
}
```

Subclass Cylinder1

```
// Cylinder1.java: Class definition for Cylinder
public class Cylinder1 extends CircleWithAccessors {
   private double length;
   public Cylinder1() {
      this (1.0, 1.0);
   }
   public Cylinder1(double r, double l) {
      super (r); // Call superclass' constructor CircleWithAccessors(r)
      length = l;
   }
   public double getLength() { return length; }
   public double findVolume() { return findArea() * length;}
}
```

Class TestCylinder

```
// TestCylinder.java: Use inheritance.
public class TestCylinder {
  public static void main(String[] args) {
    // Create a Cylinder object and display its properties
    Cylinder1 myCylinder = new Cylinder1(5.0, 2.0);
    System.out.println("The length is " + myCylinder.getLength());
    System.out.println("The radius is " + myCylinder.getRadius());
    System.out.println("The volume of the cylinder is " +
                       myCylinder.findVolume());
    System.out.println("The area of the circle is " +
                       myCylinder.findArea());
  }
}
```

subclass can just call the method that it inherited from super class

Note: getRadius() and findArea() are inherited

Using the Keyword super

The keyword super refers to the superclass of the class in which super appears. It can be used in two ways:

1. To call a superclass constructor
2. To call a superclass method.

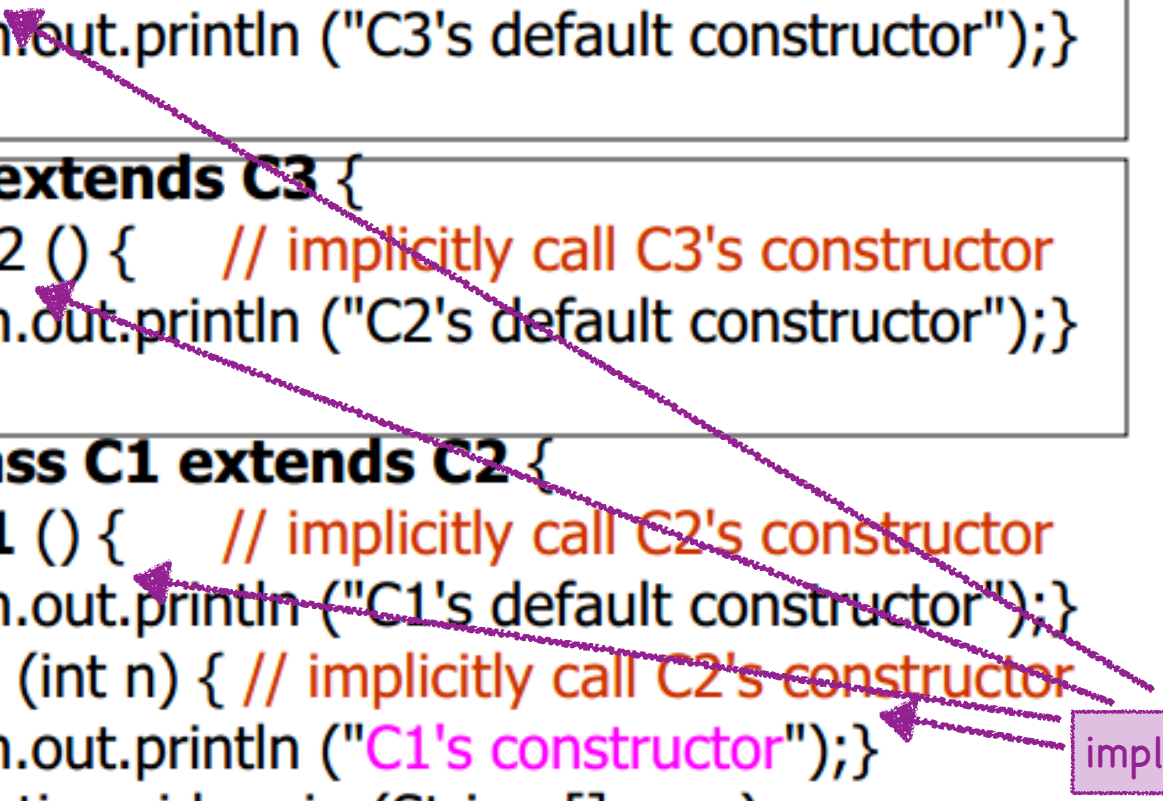
Calling Superclass Constructors

- To call a superclass constructor, use
  super () or super (parameters)
- A subclass's constructor will always invoke super () if super () or super (parameters) is not invoked explicitly in the constructor.
- The statement super () or super (parameters) must appear as the 1st line of the subclass constructor if it is called.

Examples

C3's default constructor
C2's default constructor
C1's default constructor

```
class C3 {
  public C3 () {     // constructor
    System.out.println ("C3's default constructor");}
}
class C2 extends C3 {
  public C2 () {     // implicitly call C3's constructor
    System.out.println ("C2's default constructor");}
}
public class C1 extends C2 {
 public C1 () {      // implicitly call C2's constructor
    System.out.println ("C1's default constructor");}
public C1 (int n) { // implicitly call C2's constructor
    System.out.println ("C1's constructor");}
 public static void main (String [] args)
   { new C1 (1); } // Which constructor do we call?
}
```



Superclass default constructor

If a superclass defines constructors other than a default constructor,

then the subclass cannot use the default constructor of the superclass as the superclass does not have one.

```
class B {
   public B (String name) { // non-default constructor
     System.out.println ("B's non-default constructor");
   }       // Will the compiler gives B a default constructor?
}
public class A extends B {
    // class A cannot be compiled as the default constructor
    // of A given by the compiler has a call to the default
    // constructor of B which does not exist
}
```

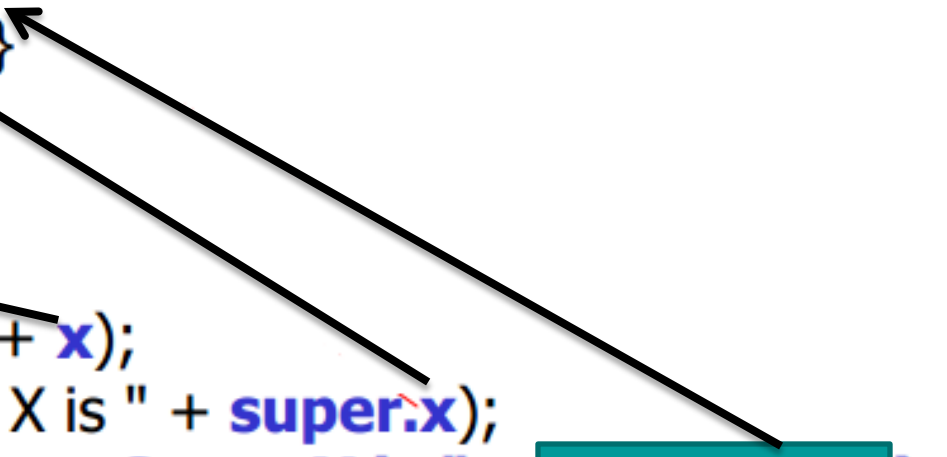
Calling Superclass Methods

- The keyword super also can be used to refer to a method other than the constructor in the superclass.
- For example, in Cylinder1 class, if it has defined its findArea() method, then to call the findArea() method in the superclass CircleWithAccessors , super is needed:

```
double findVolume () {
   return super.findArea () * length;
}
```

Accessing super-super class attributes

```
class A               { int x = 77;}
class B extends A { int x = 88;}
class C extends B {
   int x = 99;
   void printing () {
      System.out.println ("X is " + x);
      System.out.println ("Super X is " + super.x);
      System.out.println ("Super Super X is " +          );
   }
}
class SuperSuper {
   public static void main (String [] args) {
     new C ().printing ();
   }
}
```



not supposed to work around this in the spirit of encapsulation (to be verified)

Note: super.super.x is invalid.

Method Overridding

- A subclass inherits methods from a superclass.

- Sometimes it is necessary for the subclass to override the methods defined in the superclass, and this is called method overridding.

Example of method overridding (1)

```
// Cylinder2.java: New cylinder class that overrides the findArea()
public class Cylinder2 extends CircleWithAccessors {
private double length;
public Cylinder2() {length = 1.0;  }        // Where is super()?
public Cylinder2(double radius, double l) {
    super (radius);
    length = l;
  }
public double getLength() { return length;}
public double findArea() {          // method overridding
    return 2 * super.findArea() + 2 * getRadius() * Math.PI * length;
  }
public double findVolume() {return super.findArea() * length; }
}
```

super() is implicitly invoked

no, since it's inherited

Q: Do we have to specify super.getRadius()?

Example of method overridding (2)

```
// TestOverrideMethod.java: Test the Cylinder class that overrides
// its superclass's methods.
public class TestOverrideMethod {
  public static void main(String[] args) {
    Cylinder2 myCylinder = new Cylinder2(5.0, 2.0);
    System.out.println("The length is " + myCylinder.getLength());
    System.out.println("The radius is " + myCylinder.getRadius());
    System.out.println("The surface area of the cylinder is "+
                                          myCylinder.findArea());
    System.out.println("The volume of the cylinder is "+
                                          myCylinder.findVolume());
  }
}
```

Abstract Classes

- In the inheritance hierarchy, classes become more specific and concrete *with each new subclass*.

- Moving from a subclass to superclasses, the classes become more general and less specific.

super classes are more general

- When a class is *so general* that an instance cannot be created, it is *an abstract class*.

Abstract methods & abstract class

- An *abstract method* is a method *declared as abstract, not implemented, and all derived class must eventually implement*

- An *abstract class* is a class that *has at least one abstract method*

```
public abstract class GeometricObject {
  public abstract double findArea();
  public abstract double findPerimeter();
  public double semiperimeter(){
    return findPerimeter ( )/ 2;
   }
}
```

Abstract class GeometricObject

```
// GeometricObject.java:
public abstract class
GeometricObject {
  private String color = "white";
  private boolean filled;
protected GeometricObject() {}
protected GeometricObject
(String c, boolean f) {
    color = c;
    filled = f;
  }
public String getColor() {
return color;}
public void setColor(String c){
    color = c; }
```

not implemented(no content)

not implemented(no content)

```
public boolean isFilled(){return filled;}
public void setFilled(boolean f) {
    filled = f;
  }
public abstract double findArea();
public abstract double findPerimeter();
}
```

class Circle extends GeometricObject

```
public class Circle extends GeometricObject {
   private double radius;
   public Circle() {this(1.0);}
   public Circle(double radius) { this(radius, "white", false); }
   public Circle(double r, String color, boolean filled) {
      super(color, filled);
      radius = r;
   }
   public double getRadius() { return radius;}
   public void setRadius(double r) {radius = r;}
   public double findArea() { return radius*radius*Math.PI;}
   public double findPerimeter() { return 2*radius*Math.PI;}
   public boolean equals(Circle circle) {
          return radius == circle.getRadius();}
   public String toString() { return "[Circle] radius = " + radius;}
}
```

"a subclass's constructor will always invoke super() if super() or super(parameters) are not explicitly invoked in the constructor."

derived class implements the abstract method

class Rectangle extends GeometricObject

```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;
  public Rectangle() {this(1.0, 1.0);}
  public Rectangle(double width, double height) {
    this(width, height, "white", false);    }
  public Rectangle(double w, double h,
    String color, boolean filled) {
    super(color, filled);
    width = w;
    height = h;
   }
  public double getWidth() {return width;}
  public void setWidth(double w) {width = w; }
```

class Rectangle extends GeometricObject

```
public double getHeight() {return height; }
public void setHeight(double h) {height = h;}
public double findArea() { return width*height;}
public double findPerimeter() { return 2*(width + height);}
public boolean equals(Rectangle rectangle) {
    return (width == rectangle.getWidth()) &&
      (height == rectangle.getHeight());
  }
public String toString() {
    return "[Rectangle] width = " + width + " and height = " + height;
  }
}
```

derived class implements the abstract method

class Cylinder extends Circle

```
public class Cylinder extends Circle {
  private double length;
  public Cylinder() { this(1.0, 1.0);}
  public Cylinder(double radius, double length) {
    this(radius, "white", false, length);}
  public Cylinder(double radius,
    String color, boolean filled, double l) {
    super(radius, color, filled);
    length = l;
  }
  public double getLength() { return length;}
  public void setLength(double l) { length = l;}
```

class Cylinder extends Circle

```
public double findArea() {          // overridding
    return 2*super.findArea()+(2*getRadius()*Math.PI)*length;}
public double findVolume() {return super.findArea()*length;}
public boolean equals(Cylinder cylinder) {
    return (this.getRadius() == cylinder.getRadius()) &&
      (length == cylinder.getLength());
  }
public String toString() {
    return "[Cylinder] radius = " + getRadius() + " and length "
      + length;
  }
}
```

Polymorphism -- intuition

- A dog is an animal. A cat is also an animal. To describe them in classes, both Dog and Cat can be developed as subclasses of Animal class.
- All animals make noise. Given an animal (object), we can always call animal.makeNoise(). Since different animal makes different noise, the makeNoise() method in the Animal class is an abstract method.
- As a subclass of Animal, Dog has to implement makeNoise() to bark, and Cat has to implement makeNoise() to meow.
- When animal.makeNoise() is executed, polymorphism allows the correct version of makeNoise() to be called so that barking or meowing can be expected depending on whether an animal (object) is a dog or a cat.

Polymorphism

- *Polymorphism* –many forms (faces) literally.

  The ability to perform the operations according to the identity of an object instantiated from one of the many related subclasses of a class
- For example, a Cylinder, Circle, Rectangle are subclasses of GeometricObject
  - Thus a GeometricObject object has three faces. It may behave as a Cylinder, Circle, or Rectangle according to the true identity of this object
- Polymorphism can be realized through dynamic binding

Dynamic Binding

- A method may be defined in a superclass but overridden in a subclass.
- Which implementation of the method is used on a particular call will be determined dynamically by the Java Virtual Machine at runtime.
- This capability is known as *dynamic binding*, the binding of a method to its actual implementation during runtime.

Dynamic Binding

Dynamic binding works as follows:

- Suppose an object o is an instance of C_1 with C_1 a subclass of C_2, C_2 a subclass of C_3, etc. C_n is the most general class and C_1 is the most specific class.
- In Java, C_n is the Object class. If we invoke a method m() through o, the Java Virtual Machine will search for this method in C_1, C_2, …, C_{n-1} and C_n until it is found, and the first found is invoked.
- If m() is found in C_1, then it is called immediately without moving up the inheritance hierarchy. This happens when we execute o.makeNoise() with o being a Dog object of C_1, regarded as a subclass of Animal class C_2

Dynamic Binding

- Polymorphism allows methods to be used for a wide range of object arguments.
- We may pass an object as an argument of a method if the class of this object is a subclass of the class of the parameter
- The method invoked through this object is determined dynamically by the class of the argument, not by the class of the parameter.

polymorphism allows more flexible parameter-passing

(what's passed into it)

(what's required)

Example on Dynamic Binding

```
public class TestPolymorphism {
public static void main(String[] args) {
    GeometricObject geoObject1 = new Circle (5);
    GeometricObject geoObject2 = new Rectangle (5, 3);
    System.out.println("Do the two objects have the same area? "
                        + equalArea(geoObject1, geoObject2));
    displayGeometricObject(geoObject1);
    displayGeometricObject(geoObject2);
  }
static boolean equalArea( GeometricObject o1,  GeometricObject o2) {
    return o1.findArea() == o2.findArea();}
static void displayGeometricObject( GeometricObject object) {
    System.out.println();
    System.out.println(object.toString());
    System.out.println("The area is " + object.findArea());
    System.out.println("The perimeter is " + object.findPerimeter());
  }
}
```

a Circle object is passed into it

a Rectangle object is passed into it

Interfaces

- The *interface* in Java consists of *public abstract* methods and *public static final* fields only.
- A class is said to *implement* an *interface* if it provides definitions for all of the abstract methods in the *interface*
- Each interface is compiled into a separate bytecode file, just like a regular class.
- We cannot create an instance of an interface, but
- We can use an interface as a data type for a variable, as the result of casting etc.

To define an interface called *InterfaceName*, use:

```
modifier interface InterfaceName {
    /* Constant declarations */
    /* Method signatures */
}
```

casting (upcast, downcast) - turn an object into another object (through inheritance)

Implementing several Interfaces

- Sometimes it is necessary to derive a subclass from several classes, thus inheriting their data and methods. *Java*, however, *does not allow multiple inheritance*.
- The extends keyword allows only one parent class. With *interfaces*, we can achieve the effect close to that of multiple inheritance by implementing several interfaces.

  For example,

  ```
  public class Vector<E> extends AbstractList<E>
      implements List<E>, RandomAccess,
      Cloneable, Serializable
  ```

Q: What are the methods that are abstract in AbstractList<E>?

Interface Comparable<T>

- Suppose we want to design a generic method to find the larger of two objects, we can use the following interface in java.lang:

```
// Interface for comparing objects
package java.lang;
public interface Comparable<T> {
   public int compareTo (T o);
}
```

generic

- The compareTo method determines the order of "this" object with the specified object o, and returns -1, 0 or +1 if "this" object is regarded as smaller, equal, or larger than the specified object o. (Similar to the concepts of <, ==, > applicable to numbers and their wrappers)

Comparable<String>

When T is String, then the interface of interest is Comparable<String>. Any class that has implemented the method compareTo(String s) can claim to have implemented Comparable<String>.

Example:

```
class A implements Comparable<String> {
  public int compareTo(String s) { return 0;}
   public static void main (String [] args) {
     A a = new A();
     System.out.println(a.compareTo(""));
  }
}
```

Note that class A does not have anything to do with String other than having implemented the method compareTo(String s)

Using Interface As Data Type

```
public class A {
  public static Comparable <String> max(String o1, String o2) {
    if (o1.compareTo(o2 ) > 0) return o1;
    else return o2;
  }
  public static void main (String [] args) {
     String s1 = "abcdef";
     String s2 = "acdef";
     Comparable <String> s3 = max (s1, s2); // s3 supports all methods
          // described in the interface Comparable <String>
     System.out.println (s3);       // dynamic binding to toString() of
                                    // String class is done here
  }
}
```

```
[ref]
package.java.lang.*;
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Note that String class implements Comparable<String> and it is valid to return a string object as a Comparable<String> object

Interfaces vs Abstract Classes

- data
  - In an interface, all data are constants (keyword final is omitted)
  - An abstract class can have non-constant data fields.
- methods
  - In an interface, all methods are not implemented
  - An abstract class can have concrete methods.
- keyword abstract
  - In an interface, the keyword abstract in the method signature can be omitted
  - In an abstract class, it is needed for an abstract method.
- Inheritance
  - A class can implement multiple interfaces
  - A class can inherit only from one (abstract) class

Generics

Allowing operation on objects of various types

Generics: **Motivation**

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations
- In C, there is no easy way to exploit the similarity:
  - You need a separate implementation for each data type
- In Java, you can make use of **generic programming**:
  - A mechanism to specify solution without tying it down to a specific data type

Example: The Pair Class (non-generic)

- Let's define a class to:
  - Store a pair of integers, e.g. (**`74, -123`**)
  - **Many usages**: 2D coordinate, Range (min to max), Height and Weight, etc

```
class IntPair {

  private int _first, _second;

  public IntPair(int a, int b) {
    _first = a;
    _second = b;
  }

  public int getFirst() { return _first; }
  public int getSecond() { return _second; }
}
```

Usage: The Pair Class (non-generic)

```
import java.util.Scanner;

class IntPair { //definition not shown }

class TestPair {

  public static void main(String[] args) {

    IntPair range = new IntPair(-5, 20);
    Scanner sc = new Scanner(System.in);
    int input;

    do {
       System.out.printf("Enter a number in (%d to %d): ",
               range.getFirst(), range.getSecond());

        input = sc.nextInt();

    } while( input < range.getFirst() ||
               input > range.getSecond() );
  }
}
```

Sample usage of the `IntPair` class.

Purpose:
Repeatedly ask for user input until the input is in the desired range.

TestPair.java

Observation

- The **IntPair** class idea can be easily extended to other data types:
  - **double**, **String**, etc
- The resultant code would be almost the same!

```
class StringPair {

  private String _first, _second;

  public StringPair( String a, String b ) {
    _first = a;
    _second = b;
  }

  public String getFirst() { return _first; }
  public String getSecond() { return _second; }

}
```

Only differences are the data type declarations

Example: The Pair Class (Generic)

```
class Pair <T> {

  private T _first, _second;

  public Pair( T a, T b ) {
    _first = a;
    _second = b;
  }

  public T getFirst() { return _first; }
  public T getSecond() { return _second; }

}
```

The "<T>" is a formal generic type, user can **supply the desired data type later**

`T` is just like a variable, except that it stores a data type instead of a value

- Important restrictions:
  - The generic type can be substituted by **reference data type only**
    - **Primitive data types are NOT allowed**

    8 Primitive: boolean, char, (the arithmetic types:) the integral types: byte. short. int. long. (the floating-point types:) float. double.
  - Need to use Wrapper class for the primitive data type

Usage: The Pair Class (Generic)

```
class Pair <T> { //definition not shown }

class TestGenericPair {

  public static void main(String[] args) {

    Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
    Pair<String> twoStr = new Pair<String>("Turing", "Alan");

    //You can have pair of any reference data types!
    //..........
  }
}
```

TestGenericPair.java

- The formal generic type **<T>** is substituted with the actual data type supplied by the user:
  - The effect is similar to generating a new version of the **Pair** class, where **T** is substituted

Example: The Pair Class (V2.0)

- Let's modify the generic pair class such that:
  - Each pair can have two values of **different data types**

```
class Pair <S, T> {

  private S _first;
  private T _second;

  public Pair(S a, T b) {
    _first = a;
    _second = b;
  }

  public S getFirst() { return _first; }
  public T getSecond() { return _second; }

}
```

You can have multiple generic data types

Convention: Use capital single letter for the generic data type

a & b may be of diff. data type within the same object!

Usage: The Pair Class (V2.0)

```
class Pair <S,T> { //definition not shown }

class TestMoreGenericPair {

  public static void main( String[] args ) {

    Pair<String,Integer> someone =
        new Pair<String,Integer>("James Gosling", 55);

    System.out.println("Name: " + someone.getFirst());
    System.out.println("Age: " + someone.getSecond());

  }
}
```

TestMoreGenericPair.java

- This **`Pair`** class is now very flexible!
  - Can be used in many ways

Generics: **Summary**

- Caution:
  - Generics is useful when the code remains unchanged other than differences in data type
  - When you declare a generic class/method, make sure:
    - The code is valid for all possible data types
- Additional Java Generics topics (not covered):
  - Generic methods
  - Bounded generic data types
  - Wildcard generic data types