Programming in Java (23/24) – Day 8: Polymorphism

1 Polymorphism

Polymorphism is the ability of an entity to show different shapes or forms. In biology, polymorphic animals can exist in a variety of different forms, e.g., male and female animals, or different blood types. In materials science, polymorphic materials can exist in a variety of different crystalline configurations (e.g., calcite and aragonite).

In computer science, polymorphism appears at many levels. In Programming in Java, we are mainly interested in two types of polymorphism: polymorphic classes and polymorphic methods. The former is usually referred to as *generic types*, while the latter is usually called either *method overloading* or *method overriding*.

2 Method overloading

A very common form of polymorphism is *method overloading*. This is just a fancy name to say that a method can receive different parameters. We have seen several examples already, such as the method `substring` of class `String`. This method can take either one or two `int` parameters; depending on the number of arguments, the behaviour is different.

```
1        // ...
2        String str = "This is some text";
3        System.out.println(str.substring(8));     // prints "some text"
4        System.out.println(str.substring(8, 12)); // prints "some"
5        // ...
```

A method in Java is defined by its name and the types and positions of its parameters; therefore, strictly speaking, `substring(int)` is a different method from `substring(int,int)` and it is not a problem if both are present in class `String`. The following methods, however, do have the same *signature* and there can be only one of the two versions in a class:

```
1    // Same method signature! Cannot be in same class (or interface).
2    String myMethod(int first, int second, String message);
3    String myMethod(int start, int end, String txt);
```

The concept of the *signature* of a method comes up in several places in Java. The signature of a method (as a form of "unique identifier") is given by its *name* and the list of its *parameter types*. Return types or visibility modifiers are not part of the signature. In our example, the signature is:

```
1    myMethod(int, int, String)
```

This means that the names of the parameters are not important: only their types and positions. As the return type is not part of the signature either, we cannot have two methods with the same name and parameters but different return types.

```
1    // Same method signature! Cannot be in same class (or interface).
2    int myMethod(int first, int second, String message);
3    void myMethod(int start, int end, String txt);
```

When the Java compiler sees a method call, it looks at the types of the arguments and chooses which of the available *signatures* with that method name should be used. For example, in the code snippet

```
1  System.out.println("Hello, World!");
2  System.out.println(42);
```

the first method call uses a method from the Java API with signature `println(String)`, and the second method call uses `println(int)`.

Method overloading without repeating code One thing that a programmer must bear in mind with method overloading is not to repeat code. For example, the following overloaded method repeats code unnecessarily:

```
1    public void printAverage(double d1, double d2) {
2         double result = 0.5 * (d1 + d2);
3         System.out.println("The average is: " + result);
4    }
5
6    public void printAverage(double d1, double d2, String msg) {
7         double result = 0.5 * (d1 + d2);
8         System.out.println(msg + result);
9    }
```

There is repeated code among both versions of the method. In this trivial example this is just one line, but in a more serious program this could be some non-trivial algorithm that is repeated, and we know that code repetition must be avoided.

Avoiding this kind of code repetition is easy by making the more specific version of the method call the more general version, as in the code below:

```java
    public void printAverage(double d1, double d2) {
        printAverage(d1, d2, "The average is: ");
    }

    public void printAverage(double d1, double d2, String msg) {
        double result = 0.5 * (d1 + d2);
        System.out.println(msg + result);
    }
```

This is how we can have multiple polymorphic versions of the same method without any repetition of code. Remember: Don't Repeat Yourself.

Constructor overloading Also constructors can be overloaded (recall also the `Rectangle` example on Day 5). For example, we can write:

```java
/**
 * A Human has a name and an age.
 */
public class Human {

    /** This Human's name. */
    private String name;

    /** This Human's age. */
    private int age;

    /**
     * Constructs a new Human with the given name and an age of 0.
     *
     * @param name the name for this Human
     */
    public Human(String name) {
        this.name = name;
        this.age = 0;
    }

    /**
     * Constructs a new Human with the given name and age.
     *
     * @param name the name for this Human
     * @param age the age for this Human
     */
    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
32
33      /**
34       * Constructs a new Human with the same name and age as the other Human.
35       * (copy constructor)
36       *
37       * @param other the Human whose name and age will be copied to this Human
38       */
39      public Human(Human other) {
40          this.name = other.name;
41          this.age = other.age;
42      }
43  }
```

Depending on the type(s) of the argument(s), the Java compiler will again select the right constructor to use when we write `new Human(...)`.

This example has a downside: we see again repeated code. Fortunately, Java lets a constructor of a class call another constructor of the same class, using the `this(...)` notation that we saw earlier:

```
1   /**
2    * A Human has a name and an age.
3    */
4   public class Human {
5
6       /** This Human's name. */
7       private String name;
8
9       /** This Human's age. */
10      private int age;
11
12      /**
13       * Constructs a new Human with the given name and an age of 0.
14       *
15       * @param name the name for this Human
16       */
17      public Human(String name) {
18          this(name, 0);
19      }
20
21      /**
22       * Constructs a new Human with the given name and age.
23       *
24       * @param name the name for this Human
25       * @param age the age for this Human
26       */
27      public Human(String name, int age) {
28          this.name = name;
```

```
29          this.age = age;
30      }
31
32      /**
33       * Constructs a new Human with the same name and age as the other Human.
34       * (copy constructor)
35       *
36       * @param other the Human whose name and age will be copied to this Human
37       */
38      public Human(Human other) {
39          this(other.name, other.age);
40      }
41  }
```

As with the call to a superclass constructor with `super(...)`, a call to `this(...)` must come as the first statement in a constructor. We can write more statements in the same constructor after writing `this(...)`.

3 Upcasting and downcasting

Upcasting There is a second type of polymorphism in Java that is simpler but also very important. We have already seen how it works, although we have not used its proper name until now. We have been using it every time we have declared a variable by using an interface or a superclass:

```
1           Person person = new AdultPerson();
```

We know that it is good practice to work with interface names for our data types, rather than to use the class names directly. Using interfaces allows programmers to abstract themselves from all the implementation details of the class and work with just its public behaviour, i.e., the methods defined on the interface. This is called *upcasting*, and it is similar to the casting of simple types we have already seen; you just need to know that, by convention, interfaces are imagined to be "up" and specific classes are imagined to be "down" (see Figure 1), so casting from class to interface is up-casting. (The "origin" in computer science is almost always up: the root of a tree, the interface of a class, etc.)

Upcasting is a way of "moving up to know less", i.e., to intentionally forget details about the implementation so that we can work with a simpler, less specific data type. Bear in mind that a real program is really complex, and therefore reducing complexity is usually in the programmer's benefit; that is why upcasting is so common in Java programs.

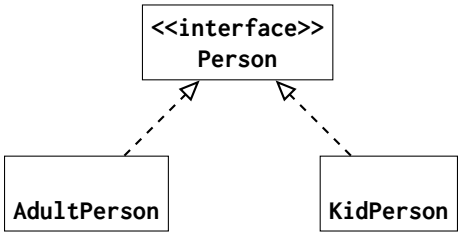

Figure 1: Interface `Person` is implemented by classes `KidPerson` and `AdultPerson`. By convention, the interface is represented "up" and the classes are represented "down". In a class diagram like this, an arrow with a dashed line and with a white triangular arrow tip means that the class at the beginning of the arrow implements the interface at the end of the arrow. Interfaces are usually labelled with "**<<interface>>**".

The concept behind upcasting is called Liskov's *substitution principle*, named after Turing Award winner Barbara Liskov.[1] The principle says for Java:

A subclass reference can be used where a superclass reference is expected.

The substitution principle works for interface types in the same way as for classes. This is why we can use a `KidPerson` reference where a `Person` reference is expected. On Day 6, we saw a good example for this. When we compile method

```
1    public static void moveAndMakeSpecialStatement(Person child) {
2        // move in front of parent
3        child.move(10);
4        // give the message
5        child.makeVerbalStatement("I love you.");
6    }
```

the compiler *does not know* what code will be used for the method calls to the methods `move(int)` and `makeVerbalStatement(String)`. When the method is called from another class, the formal parameter `child` might be replaced with an actual parameter of type `KidPerson` or `AdultPerson` or some other class that implements the `Person` interface. That other class might not even exist when method `moveAndMakeSpecialStatement(Person)` is compiled (maybe next year someone writes a class `RobotPerson` that implements interface `Person`). So when we write the method, we do not know what "shape" the object referenced by `child` will have.

This is perfectly fine for the Java compiler. It checks only that the type `Person` has suitable method *signatures* for the calls to the methods. It needs to know only *that* there will be some method implementation if an object of type `Person` is ever constructed and passed to the method. It does *not* need to know *how* that method will work, or where exactly it can be found.

Only much later, when the program is run, must the Java Virtual Machine choose which *implementation* of the method signatures to use. The JVM has much more information than the compiler: it sees what objects are actually in the computer's memory, and it knows to which classes they belong. This

[1] The Turing Award is sometimes called "the Nobel Prize of Computer Science".

allows it to pick the implementations of `move(int)` and `makeVerbalStatement(String)` in the right classes (either overridden or inherited).

This kind of polymorphism is also known as *subtype polymorphism* (because method implementations of a subtype may be used) or *ad-hoc polymorphism* (because it is decided only on-the-fly, when we run the program, what method implementation will be used).

Downcasting In some rare occasions, programmers need to do the opposite, i.e., move down from the general to the specific: this is called *downcasting*. The syntax in this case is very similar to the casting of simple types. Imagine that we had an array of animals, but we knew (somehow) that all those animals were dogs, and we wanted to make them bark. As `bark()` is a method of `Dog` but not of `Animal`, we would need to downcast them:

```
1    public void makeThemBark(Animal[] animals) {
2        for (int i = 0; i < animals.length; i++) {
3            Animal nextAnimal = animals[i];
4            Dog dog = (Dog) nextAnimal;
5            dog.bark();
6        }
7    }
```

Upcasting is an operation that is checked for errors at compile time: if the code compiles, it will always run. This is not true of downcasting, which can fail at *runtime*, even if it compiled successfully. In the example above, the method `makeThemBark(...)` may receive an array of `Animal` where not all the elements are really objects of class `Dog` (maybe there is a `Cat`). When the code tries to downcast a class into something that it is not, Java will throw a `ClassCastException`:

```
java.lang.ClassCastException: Cat cannot be cast to Dog
```

Java provides us with a special operator that allows us to check whether an object is an instance of a specific type. This operator is called **`instanceof`**. In our example, we could use it as follows:

```
1    public void makeThemBark(Animal[] animals) {
2        for (int i = 0; i < animals.length; i++) {
3            Animal nextAnimal = animals[i];
4            if (nextAnimal instanceof Dog) {
5                Dog dog = (Dog) nextAnimal;
6                dog.bark();
7            }
8        }
9    }
```

If `nextAnimal` actually is a reference to a `Dog` object (or a subclass of `Dog`, say `Labrador`), then the condition of the **`if`**-statement will become true, and we know that it is safe to downcast the reference `nextAnimal` from type `Animal` to type `Dog`. If `nextAnimal` is not a reference to a `Dog`, the condition will become false, and we will not accidentally try to make a `Cat` bark.

Since Java 16, we can write the same code also a bit shorter with *pattern matching*:

```
1    public void makeThemBark(Animal[] animals) {
2        for (int i = 0; i < animals.length; i++) {
3            Animal nextAnimal = animals[i];
4            if (nextAnimal instanceof Dog dog) {
5                dog.bark();
6            }
7        }
8    }
```

Here the type check for `nextAnimal` and, if the check evaluates to `true`, also the downcast into the variable dog are done in the same line.[2]

When we override method `equals(Object)` from class `Object` in our own classes, we usually need a downcast. Recall class `Human` with member variables `name` and `age`. Imagine that a `Human` should be equal to another object if the other object is also a reference to an instance of class `Human` and both have the same content in their `name` variables (but we ignore their age variables):

```
1    @Override
2    public boolean equals(Object o) {
3        if (! (o instanceof Human)) { // also deals with o == null
4            return false;
5        }
6        if (this == o) { // same objects always have the same content
7            return true;
8        }
9        Human other = (Human) o;
10       if (this.name == null) { // either both name variables are null...
11           return other.name == null;
12       }
13       // ...or both are references to Strings with the same content
14       return this.name.equals(other.name); // equals(null) should always be false
15   }
```

We may be tempted to write in line 11 instead:

```
11           return o.name == null;
```

However, the compiler knows only that `o` is a reference of type `Object`, and type `Object` does not have a field called `name` that we could access here. This is why it would refuse to compile the code with an error message. We need to use a reference of type `Human` to access the fields of class `Human`.

[2] This is similar to pattern matching in many functional programming languages.

4 Generic types/classes

A generic type or generic class is a complex type that can be combined with other different types to produce a new type. The most common application of generic types is for "container" classes like lists, stacks, queues, sets, and maps.

To explain the concepts, we will consider perhaps the simplest container there is: a "box" object that can store a single reference to another object and return that reference. We will soon see that the same principles apply also to much more useful containers that can store many elements (lists, stacks, queues, ...).

A first attempt to write a class that can store a single reference to an object may look as follows:

```
1  /**
2   * An OldBox stores a reference to an object and can return the stored reference.
3   */
4  public class OldBox {
5      /** The stored reference. */
6      private Object data;
7
8      /**
9       * Constructs an OldBox that stores the given reference.
10      *
11      * @param data the reference to store
12      */
13     public OldBox(Object data) {
14         this.data = data;
15     }
16
17     /**
18      * Returns the stored reference.
19      *
20      * @return the stored reference
21      */
22     public Object getData() {
23         return this.data;
24     }
25 }
```

We might then use the class as follows, say in a class `OldBoxDriver`:

```
1          OldBox intBox = new OldBox(42);
2          int x = (Integer) intBox.getData(); // ok
3          OldBox strBox = new OldBox("Hello");
4          String s = (String) strBox.getData(); // ok
5          intBox = strBox; // (!)
6          int y = (Integer) strBox.getData(); // (!)
```

This code compiles without problems. When we run the code, lines 1–4 work just as expected, even though having to write the explicit downcasts is perhaps a bit inconvenient.

Line 5 works as well, without any problems: the JVM does not know that `intBox` is intended for storing integer values, so making `intBox` point to the `OldBox` object with a reference to a `String` object is accepted.

Line 6 then leads to a `ClassCastException`. Here we are trying to downcast the `Object` reference returned by `strBox` not to a `String` (which would work), but to an `Integer` (which does not work – `String` is not a subclass of `Integer`).

Thus, `OldBox` is very flexible in what it accepts (it accepts objects of all classes), but when we retrieve the stored reference, its type `Object` tells us nothing about the actual type of the content of the `OldBox`. The problem is that a downcast instruction is inherently dangerous if we do not know for sure (e.g., because we have checked with **`instanceof`**) that the reference actually points to an object of the right type.

It would be much safer to have specialised box classes for each type that we may want to store (omitting the JavaDoc here – it would look very similar in each case anyway):

```
1 public class IntegerBox {
2     private Integer data;
3     public IntegerBox(Integer data) {
4         this.data = data;
5     }
6     public Integer getData() {
7         return this.data;
8     }
9 }
```

```
1 public class StringBox {
2     private String data;
3     public StringBox(String data) {
4         this.data = data;
5     }
6     public String getData() {
7         return this.data;
8     }
9 }
```

```
1 public class PersonBox {
2     private Person data;
3     public PersonBox(Person data) {
4         this.data = data;
5     }
6     public Person getData() {
7         return this.data;
8     }
```

```
}
```

... and so on for all classes and interfaces whose objects we may want to put into our container. This is useful: now lines 5 and 6 in our code that uses `IntegerBox` and `StringBox` are rejected by the compiler:

```
        IntegerBox intBox = new IntegerBox(42);
        int x = intBox.getData(); // No cast needed!
        StringBox strBox = new StringBox("Hello");
        String s = strBox.getData(); // No cast needed!
        intBox = strBox; // Rejected by the compiler.
        int y = (Integer) strBox.getData(); // Rejected by the compiler.
```

But this is very much not in line with the running theme of our module: *Don't Repeat Yourself*. The three classes `IntegerBox`, `StringBox`, and `PersonBox` (and all their variations that we may write) look almost exactly the same – the only thing that changes from one to the other, apart from the class name, is the *type* of the data that we accept for storage and later for retrieval with the same type.

It would be marvellous if we could write our class just once, as with `OldBox`, but still getting the type safety that comes with all the many different copies that we wrote in the second attempt. Then we would not have to repeat ourselves, and the compiler would still be able to prevent type errors that would otherwise surface at runtime. Since Java 5 (introduced in 2004), this is possible with *generic types*.

4.1 Writing and using generic classes

A generic class is very similar to a normal class. The only difference – but it is a crucial one – is that it uses one or more *type parameters* to indicate that it needs another type to be fully declared. Type parameters are specified in angle brackets ("<", ">").

Let's see a generic version of our container class:

```
/**
 * A Box<T> stores a reference to an object and can return the stored reference.
 *
 * @param <T> the type of the reference to store and later retrieve
 */
public class Box<T> {
    /** The stored reference. */
    private T data;

    /**
     * Constructs a Box<T> that stores the given reference.
     *
     * @param data the reference to store
     */
    public Box(T data) {
```

```
16          this.data = data;
17      }
18
19      /**
20       * Returns the stored reference.
21       *
22       * @return the stored reference
23       */
24      public T getData() {
25          return this.data;
26      }
27  }
```

The type parameter T represents a type to be determined when the class Box<T> is used. For example, we could declare a Box<Integer> or a Box<String> (see below).

The way to pronounce a type like Box<T> is "Box of T", and similarly, Box<String> is "Box of String".

As you can see, there are two main differences between a normal class and a generic class (and their methods):

- The name of the class is extended to include the type, by using a type parameter in "< >". Capital T (for "type") is a common name for type parameters. By convention, type parameters are usually capital single letters (T for "type", K for "key", V for "value", E for "element", etc), but any valid identifier can be used.
- The variable type can be used in every place where a type (simple or complex) should be: parameter types, return types, and for declaring new variables.

How are generic classes (and interfaces) used? Look at this code:

```
1          Box<Integer> intBox = new Box<Integer>(42);
2          int x = intBox.getData(); // No cast needed!
3          Box<String> strBox = new Box<String>("Hello");
4          String s = strBox.getData(); // No cast needed!
5          intBox = strBox; // Rejected by the compiler.
6          int y = (Integer) strBox.getData(); // Rejected by the compiler.
```

As you can see, the same generic class can be used to handle different specific classes, i.e., boxes of integers and strings in this case. Generic types must be based on complex types (classes or interfaces), not simple types. Boxed types are useful if you need a generic type based on one of the primitive (i.e., simple) types. This is why the example uses a Box<Integer>; if you try to create a Box<int>, the Java compiler will complain.

So with our generic class Box<T>, we get all the benefits of the approach with IntegerBox, StringBox, PersonBox, ... for compile-time checking of type safety (no ClassCastException at runtime) without the drawback of repeating code.

Since Java 7, we can use a shorthand notation for calling constructors of specific generic types: instead of

```
1        Box<Integer> intBox = new Box<Integer>(42);
2        Box<String> strBox = new Box<String>("Hello");
```

we can write:

```
1        Box<Integer> intBox = new Box<>(42);
2        Box<String> strBox = new Box<>("Hello");
```

When the Java compiler comes across the call to `new Box<>()` here, it looks up what type should be "filled in" between the <> from the type of the variable in the left-hand side of the assignment. In this example, variable `intBox` has type `Box<Integer>`, so the compiler knows that here `new Box<>()` is shorthand for `new Box<Integer>()`. Similarly, `strBox` has type `Box<String>`, so the compiler knows that here `new Box<>()` is shorthand for `new Box<String>()`.

However, it is unsafe just to omit the <> altogether — that would tell the compiler not to do *any* type checks for the element types and to use the "raw type" `Box` without information about the element type instead of `Box<Integer>` or `Box<String>`. The Java compiler accepts such code by default for "backwards compatibility", to ensure that the code still works with Java code that was written before generics types were introduced with Java 5 in 2004. If we know that we are not working with such "legacy code", it is sensible to make the compiler stricter and reject code that does not use generic types where they are needed.[3]

You should always use generic types correctly — certainly in the *Programming in Java* module — and never use raw types, unless there is no alternative because of legacy code.

Note that when we have an object `myPersonBox` of type `Box<Person>`, it means that `myPersonBox` can store references to objects of classes that implement the `Person` interface (e.g., `KidPerson`, `AdultPerson`, etc). The substitution principle applies here as well.

4.2 Restricting generic types

There are situations in which you want to create a generic type that can only be combined with some specific types. For example, you may want to have a special kind of box, a `MathBox` that can do mathematical operations on its content. Of course, that would work only if the content is some kind of number. On Java level, we want to be able to call the method `double doubleValue()` that is available in the abstract class `Number` in the Java libraries. This abstract class has subclasses like `Double`, `Integer`, `Long`, and `BigInteger` (for *unbounded* integer numbers).

Our first attempt may look as follows:

[3] To make the Java compiler reject code that does not use generics properly, you can write:
`javac -Werror -Xlint:rawtypes MyClass.java`
If you are using the IntelliJ IDE, you can open the "Code Inspections" configuration panel (see `https://www.jetbrains.com/help/idea/code-inspection.html`), then select "Java/Java language level migration aids/Java 5/Raw use of parameterised class" and switch "Warning" to "Error".
If you are using the Eclipse IDE, go via the menu "Window" → "Preferences", then go to "Java" → "Compiler" → "Errors/Warnings", then go to "Generic Types" → "Usage of a raw type" and switch "Warning" to "Error".

```java
public class MathBox<T extends Number> {
    private T data;

    public MathBox(T data) {
        this.data = data;
    }

    public T getData() {
        return this.data;
    }

    /**
     * @return the square root of the encapsulated number
     */
    public double sqrt() {
        double d = this.data.doubleValue();
        return Math.sqrt(d);
    }
}
```

Note that the keyword to restrict the type parameter is always **extends**, not **implements**, even if the type used for the restriction is an interface type.

We can use class `MathBox<T extends Number>` as follows in our programs:

```java
MathBox<Integer> intBox = new MathBox<>(5); // ok: 5 is auto-boxed into Integer
MathBox<Double> doubleBox = new MathBox<>(32.1); // ok: 32.1 is auto-boxed into Double
MathBox<String> strBox = new MathBox<>("No good!"); // compile-term error
```

But here we are repeating code that is already present in class Box. Fortunately, generic classes also support writing subclasses:

```java
public class ImprovedMathBox<T extends Number> extends Box<T> {
    public ImprovedMathBox(T data) {
        super(data);
    }

    /**
     * @return the square root of the encapsulated number
     */
    public double sqrt() {
        Number data = this.getData();
        double val = data.doubleValue();
        return Math.sqrt(val);
    }
}
```

4.3 Method-scoped generic types

We can use methods with generic types also in classes that are not generic themselves. A (somewhat artificial) example:

```
1  public class BoxWrapper {
2      public static <T> Box<T> wrapInBox(T data) {
3          return new Box<>(data);
4      }
5      public static void printGreeting() {
6          System.out.println("Greetings from the BoxWrapper!");
7      }
8      // ... more methods ...
9  }
```

Here the class `BoxWrapper` itself is not generic: in general, it is good practice to avoid introducing generics where they are not needed. Instead, by writing `<T>` before the return type of method `wrapInBox`, we tell the Java compiler that in method `wrapInBox`, the name `T` stands for a variable that represents a type (rather than an actual class or interface that might be found in a file `T.java`). Outside of method `wrapInBox`, for example in method `printGreeting`, the type `T` would not be recognised (unless you happen to have a class or interface `T` in the same package or in your imports).

4.4 Conclusion

Generic classes are one of the main types of polymorphism present in Java. They enable programmers to combine one class with several other classes, by using type parameters.

Generic classes are, therefore, a way of reusing code and/or avoiding code repetition (remember the DRY principle: Don't Repeat Yourself). There is no need to create a list of integers, a list of strings, and a list of books. It is better to create a generic list that can be combined with any type, and then combine it with integers, string, books, or any other type as needed. The drawbacks of generic classes is that they are slightly more verbose (i.e., the programmer needs to write more) and that they can become very confusing if abused, e.g., when generic classes of generic classes of generic classes are employed.

In practice, it is somewhat more common to use existing generic classes and provide the missing type information than to write generic classes ourselves. We will soon have a look at the Java *Collections framework*, which contains interfaces and implementations of several kinds of container classes and makes heavy use of generics. In this context, we will also look at *wildcards* as a mechanism to use generic types a bit more flexibly.

A full description of generics in Java, how exactly they are implemented internally, and the full list of possibilities and caveats, is beyond the scope of this course.[4] As a rule of thumb, remember these simple rules:

- You cannot create new objects of generic types. Code like `new T()` or `new T[]` will generate a compile error.
- You cannot create local variables of generic types that have not been declared between "< >" for the class or method. Code like `T newNumber = new Integer(1);` will generate a compile error.

And remember these two pieces of advice:

- Use generic *sparingly* in your Java code, and always with good reason.
- Do not abuse generics creating generic types of generic types (your brain will thank you in the long term).

[4] The module *Software Design and Programming* on the MSc Computer Science at Birkbeck will revisit generics and provide more information.