Generic Classes and Generic Methods

Intro to Generic Classes

> **Generic class**: A generic class is a parameterized class where the parameters are always object types.

```
// T1, T2, ... are object type parameters
public class ClassName<T1, T2, ...> {
    // we can use T1, T2, ... as type specifier here
}
```

```
public class GenericStroe<T> {// T is the type parameter
    private T data; // variable of the T type
    public GenericStore(T data) {
        this.data = data;
    }
    public T getData() { // return T type variable
        return this.data;
    }
}
```

- The Java compiler will remember the places where the generic type parameter `T` are used.
- The generic type parameter `T` tells the Java compiler to place the correct cast operation at some result before using it.
- When the parameter type is `<T>`, the Java compiler will replace every occurrence of `<T>` by `Object`.
- When we define a variable of a generic class, we specify the object type parameter along with the class name:

  ```
  GenericsStore<String> a = new GenericsStore<String>();
  GenericsStore<Integer> b = new GenericsStore<Integer>();
  ```

- The result will be:
  - The Java compiler will remember the parameter type of each variable, and
  - Insert the proper casting operation before using the value returned by their methods.

- The parameters of a generic class must be object (reference) types
  - We cannot define generic class variables using primitive types:

```
GenericsStore<int> a = new GenericsStore<int>(); // Illegal!
```

  - Use a wrapper class if we need to use a primitive type.
- We can use a short hand notation to define a generic class variable:

```
GenericsStore<Integer> a = new GenericsStore<>();
```

  - The Java compiler can infer the second parameter
- Commonly parameter names used are: `T` (Type), `E` (Element), `K` (Key), and `V` (Value)

Intro to Generic Methods

- Syntax to define a generic (parameterized) method

```
public static <T1, T2, ...> returnType methodName(params) {
  // method body
}
```

  - We can use the type parameters `T1, T1, ...` to declare parameter variables, local variables, and the return type of the method.

```
public static <T> void print(T[] list) {
  for (int i = 0; i < list.length; i++) {
      System.out.println(list[i]);
  }
}
```

  - The generic method will be made specific (with a data type) in the invocation.
- When we write a generic method, the Java compiler will replace every occurrence of `<T>` by `Object` and will remember that `list[]` is a parameterized class variable.
  - When the `print()` method is used, the Java compiler will insert the appropriate casting operator.
- Syntax to invoke a generic (parameterized) method:

```
Classname.<T1, T2, ...>methodName(arguments)
```

Bounded and Unbounded Parameter Type

- An unbounded generic type parameter `T` is specified as `<T>` or `<T extends Objects>`
  - We can use any object (reference) type to make the parameter type `T` into a specific type.
  - When an unbounded generic type parameter `T` is used in a generic class definition, the type parameter `<T>` is replaced by `Object`.
- When `Object` is inappropriate as the parent class: `Object` does not have certain required methods used in the code.
- A bounded generic type parameter `T` is specified as `<T extends SuperClass>`
  - In this way, we can only use a subtype of a superclass to make `T` into a specific type.
  - When a bounded generic type parameter `T` is used in a generic class definition, the type parameter `<T>` is replaced by the bounding type, instead of `Object`.
- The use of bounded type parameter is necessary when we have used a method in the code that is not defined in the `Object` class.