Generic Methods and Classes

Weiss, sections 4.2 through 4.5
(pages 125–141)
Why Use Generics

• Runtime bugs are hard to detect. Even after detecting, bugs do not indicate the actual cause of a problem.
• Generics make bugs detectable at compile time.
• Type parameters allows reuse the same code with different parameters.
• Generics are types (e.g., classes). Think of generics as the parameters of a method. The method body executes with different types.
Non-generic Type

- Consider a non-generic `Box` class
- Box methods accept or return an object.
- Some code passes an `Integer` to `set` and expects an integer from `get`. Another code mistakenly passes a string (runtime error).

```java
public class Box {
    private Object object;

    public void set(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}

public class TestBox {
    public static void main(String[] args) {
        Box box = new Box();
        box.set(10);
        Integer a = (Integer)box.get();

        box.set("Hello");
        Integer b = (Integer)box.get();
    }
}
```

Runtime error
Generic Types

• Define **type parameter section** after class declaration. It is delimited by <> and consists of type parameters/variables e.g., T.
• A type variable can be any **non-primitive** type.
• A generic class can have **multiple** type parameters. Type parameters are separated by **comma**.

```java
public class Box<T> {  
    // T stands for Type  
    private T t;

    public void set(T t) {  
        this.t = t;
    }

    public T get() {  
        return t;
    }
}
```
Generic Types

• To instantiate a generic class, replace generic type with a **concrete** type.
• **Integer** and **String** are called **type arguments**.
• **No cast** is required for the generic classes.

```java
public class TestBox {
    
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(10);
        Integer a = integerBox.get();

        Box<String> stringBox = new Box<String>();
        stringBox.set("Hello");
        String b = stringBox.get();

        stringBox.set(2) does not compile.
    }
}
```
Raw Type

• Raw (non-generic) type is the name of a generic class without type arguments.

Box rawBox = new Box();

• Raw type appear in classes that were not generic before Java 5.
• For backward compatibility, it is allowed to assign a parameterized type to a raw type.

Box<String> stringBox = new Box<String>();
Box rawBox = stringBox;

• Assigning a raw type to a parameterized type generates a warning.
Box rawBox = new Box();
Box<Integer> integerBox = rawBox; // warning: unchecked conversion
Generic Method

• For a generic method, type parameter section appears before the return type of the method.
• Use a type parameter within the scope of the method where it is declared.

```java
public static <T> void printBoxArray(Box<T>[] boxArray) {
    for(Box<T> element : boxArray) {
        System.out.println(element.get());
    }
}
```

Box<Integer>[] boxArray = new Box<Integer>[5];  Cannot create an array of generic objects

Box[] boxArray = (Box[]) new Object[5];  Create an array of Object and then downcast
Box.printBoxArray(boxArray);
Bounded Type Parameters

- Restrict types that are used as type arguments.
- Use `extends` keyword followed by an upper bound.

```java
public <U extends Number> void inspect(U u) {
    System.out.println(u);
}
```

```java
Box<Integer> integerBox = new Box<Integer>();
integerBox.set(10);
integerBox.inspect("some text"); // inspect accepts numeric arguments.
```
Inheritance and Subtypes

• Type compatibility applies to generics.

```java
Box<Number> box = new Box<Number>();
box.set(new Integer(10));
box.set(new Double(10.1));
```

• You can not pass Box<Integer> or Box<Double> to boxTest.
• Box<Integer> and Box<Double> are not subtypes of Box<Number>.
• The common parent of Box<Integer> and Box<Double> is `Object`.

```java
public void boxTest(Box<Number> n) {
}
```
Inheritance and Subtypes

• You can subtype a generic class by extending or implementing it.
• `ArrayList<T>` is a subtype of `List<T>` because `ArrayList<T>` implements `List<T>`.
• `List<T>` is a subtype of `Collection<T>` because `List<T>` extends `Collection<T>`.

⚠️ The type arguments need to be the same.
### Wildcard

- In generic code, the question mark (?) is called wildcard.
- It indicates an unknown type.
- Upper bound wildcard relaxes the type of the unknown type to be a specific type or a subtype of that specific type.
- A method that accepts a list of Number and subtypes of Number.

List<? extends Number> is more relaxed than List<Number>

```java
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for(Number n : list) {s += n.doubleValue();}
    return s;
}
```
Wildcard

- Lower bound wildcard restricts an unknown type to be a specific type or a super type of that specific type.

- A method that stores Integers in a list. You can make it flexible to store a list Numbers or Objects too.

- List<? super Integer> is more relaxed than List<Integer>.

```java
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i < 10; i++) {
        list.add(i);
    }
}
```
Wildcards and Subtyping

- Integer is a subtype of Number.

```java
List<Integer> intList = new ArrayList<Integer>();
List<Number> numList = intList; // compile-time error
```

- The common parent of List<Integer> and List<Number> is List<?>.
- Use upper bound wildcard to create a relationship between List<Integer> and List<Number>.

```java
List<Integer> intList = new ArrayList<Integer>();
List<? extends Number> numList = intList;
```