Generic Classes

- A class that operates on different data types

→ Type Parameter (T) is a placeholder for the type. When you create an instance, you specify the actual type

- Syntax: angle brackets < > after class or method name

1) Example

```
Public class Box<T> {
    private T content;
    public void put(T item) {
        content = item;
    }
    public T get() {
        return content;
    }
}
```

→ Using the Generic Class

```
Int Box.put(42);          → so T item type: Integer
StringBox.put("Hello");
```

```
Int intValue = intBox.get();
Strin stringValue = stringBox.get();
```

↳ Gets the int value so wrapper → Primitive

* Could have pairs *

```
class Pair<T1, T2> { ... }
```

```
Pair<Integer, String> ... = new Pair<>(42, "Hello");
```

2 Parameters

Creating An Instance

```
Box<Integer> intBox = new Box<>();
Box<String> stringBox = new Box<>();
```

* Wrapper class *

—can't directly use primitive types as type parameters—

WRAPPER CLASSES

- Wrapper classes are classes that encapsulate primitive types (8) within an object

→ int → Integer
→ double → Double
→ char → Character

- When 2 methods need to refer to the same instance of a primitive type, pass a wrapper class as a method argument

* Say you have 2 methods — Method A and Method B — that need to work with the same integer value. So they need to refer to the same instance of that integer. So pass an Integer (wrapper of int) so that way both methods operate on the same Integer object, sharing its state *

- Primitive to Wrapper:

```
Integer obj = new Integer(10);
```

- Wrapper to Primitive

```
Integer obj = new Integer(10);
int val = obj.intValue();
```

1) Autoboxing — automatic conversion of primitive type → wrapper class

```
Integer wrappedInt = 10;
```

2) Unboxing — vice versa

```
int primitiveInt = wrappedInt;
```