An abstract data type defines a type by defining the methods you can run on it

**Abstract data type**: defined by behavior

**Concrete data type**: defined by representation

Defining a type in terms of its behavior gives us representation independence – we can change the representation, with no visible effect to the user/client

**Stack**

Methods:
- Empty()
- Push(item)
- Pop()
- Peek()
- Size()
- Equals(stack)
- Print()
- Get(i)

There are a lot of different methods we could define on a stack. To define our ADT, we'll use the four in bold.

In addition, our ADT is going to represent an immutable stack.

```java
public interface Stack<T> {
    public boolean empty();
    public Stack<T> push(T item);
    public Stack<T> pop();
    public T peek();
    public Stack<T> equals(stack);
    public int size();
    public void print();
    public T get(i);
    ...
}
```

```java
public class RecursiveStack<T> implements Stack<T> {
    private T top;
    private Stack<T> rest;
    ...
}
```

```java
public class ListStack<T> implements Stack<T> {
    private List<T> contents;
    ...
}
```

**BigInteger**

Methods:
- +, -, /, *, ==, ^
- getVal(String)
- getVal(int)

```java
BigInteger
```

Representations:
- List of bits plus sign flag: -4 = {1, 0, 0}; sign = -1
- String: -4 = "-4"
- List of integers

```java
class BitBigInteger implements BigInteger {
    private boolean[] bits;
    private int sign;
    ...
}
```

```java
class IntBigInteger implements BigInteger {
    private int[] bits;
    private int sign;
    ...
}
```

**Stack**

Representations:
- java.util.List
- Cons/Cdr
- Array
- Linked List
- Recursive structure

```java
public class RecursiveStack<T> implements Stack<T> {
    private T top;
    private Stack<T> rest;
    ...
}
```

```java
public class ListStack<T> implements Stack<T> {
    private List<T> contents;
    ...
}
```

**BigInteger**

Representations:
- List of bits plus sign flag: -4 = {1, 0, 0}; sign = -1
- String: -4 = "-4"
- List of integers

```java
class BitBigInteger implements BigInteger {
    private boolean[] bits;
    private int sign;
    ...
}
```

```java
class IntBigInteger implements BigInteger {
    private int[] bits;
    private int sign;
    ...
}
```

\[
x = \sum_{i=0}^{k} bits[i] \times N^{k-i}
\]

\[N = \text{Integer.MAXVALUE} + 1\]
Once we've defined an abstract data type and we've picked a particular representation, it's possible that the representation we chose is a super set of the data types our ADT represents.

A rep invariant holds when the current state of the representation represents a valid type of our ADT.

Good design to write a method checkRep() which we call to see if our invariant still holds – won't be able to recover if there's an error, but useful for debugging.

**Tic Tac Toe**

*Move = {X, O, None}*

**Abstract Data Type**

Methods:
- `getVal(r,c) → Move`
- `doMove(r,c,move)`

**Representations**

- 2D array of moves
- Two lists of coordinates – one contains list of moves by X player, other contains moves by O player

---

`==` checks pointer equality

- do both objects point to the same location in memory?

`equals(...)` does whatever we tell it to do

- You should override it to implement a useful equality test
- If you don't override it, default implementation is `==`
- If you do override it, you should make sure:
  - `hashCode()` returns same value for equal objects
  - Reflexive: `x.equals(x)`
  - Symmetric: `x.equals(y) <=> y.equals(x)`
  - Transitive: `x.equals(y) and y.equals(z) => x.equals(z)`
  - Consistent across calls

---

Each time we call “new” we’ll create a new object in memory (`==` will evaluate to false), even if those objects contain the same data.

Interner is a method of reusing immutable objects, so that two “equivalent” objects are really the same object in memory. This is useful because:
- It means that `==` implies “equivalent” objects
- Saves space