1.
```
function returnFirst<T>(items: T[]): T{
    return items[0]
}
console.log(returnFirst<number>([4,2,1]));
```

2. number

3.
```
function pair<T, U>(a: T, b: U): string {
    return String(a)+String(b)
}
```

4.
```
function    identity<T>(value: T): T {
    return value
}
console.log(identity<boolean>(true));
```

5. The type will be inferred if we do not specify type when calling the function.

6.
```
function    logItems<T>(items: T[]): void {
    items.forEach(x=>console.log(x))
}
logItems<number>([2,3,4,5,6])
```

7. a
   b
   c

8.
```
let arr:Array<number>;
arr=[10, 20, 30]
```

9.
```
function displayStrings<T>(arr: T[]): void {
    arr.forEach(str => console.log(str));
}

displayStrings<string>(["asd","fer","uir"])
```

10. Yes

```
function demo<T>(a:T[]):void{
    console.log(a);
}
demo([{name:"asd"},{name:"ddsd"},{name:"try"}])
```

11.
```
interface Box<T>{
```

```
    value:T
}

const b1:Box<string>={value:"Box"}
```

12.
```
interface Box<T,U>{
    name: T,
    age:U
}
const userBox:Box< string, number>={name:"Abhi",age:20}
```

13.
```
interface  ApiResponse<T> {
    data:T,
    success:boolean
}
```

14.
```
interface Wrapper<T> {
 item: T;
 }
 let wrapped: Wrapper<string> = { item: 'hello' };
```

15. This is not valid because the Container inference specifies a number type using a generic, but the property value is assigned a different type.

16.
```
class Stack<T>{
    constructor(public arr:T[]){}
    push(a:T){
        this.arr.push(a)
    }
    pop(){
        this.arr.pop();
    }
}

const s1:Stack<string>=new Stack<string>(["alan","abhi"])
s1.push("arun");
console.log(s1.arr);
s1.pop();
console.log(s1.arr);
```

17. 123

18. Yes.
```
class Container<T,U> {
```

```
constructor(private value1: T,private value2: U) {}
getValue(): U {
console.log(this.value1);
return this.value2;
}
}
const c = new Container<number,string>(123,"anbu");
console.log(c.getValue());
```

19.
```
class  Queue<T>{
   constructor(public items:T[]){}
   enqueue(item: T) {
      this.items.push(item)
   }
   dequeue(): T | undefined {
      return this.items.shift()
   }
}
```

20. Generics are useful in class design because it enables us to create classes in a generic manner in which we could specify their type at the time of object creation.

21.
```
function    getLength<T extends {length:number}>(item: T): number{
   return item.length
}
console.log(getLength("jjhhjh"));
```

22. A problem will generate because we pass a number type as an argument. Number type do not have length property.

23.
```
function sayName<T extends {name:string}>(obj: T): string {
   return obj.name;
}
```

24.
```
function     merge<T extends object, U extends object>(a: T, b: U): T & U{
   return {...a,...b}
}
const a={name:"Zera"}
const b={name:"Mathew"}
console.log(merge(a,b));
```

25.
```
function    wrap<T = string>(value: T): T[] {
   let arr:T[]=[];
   arr.push(value);
```

```
    return arr
}
console.log(wrap("12333"));
```

26. [42]

27.
```
function checker<T=boolean>(success:T):void{
    console.log(success);
}
checker(true)
```

28.
```
function parse<T=number>(input: T): string {
        return String(input);
}
```

29. Two advantages :-
    - Preserve type information
    - Ensure type safety

30. By using the first method the type which will be assigned or inferred to T and it must be used all across the function where the type is specified T like for value and return type. But in the second case we could accept any value to variable value and the return type should not need to be the type of value.