CHAPTER 3

CONCEPTS

3.1 Variables and lifetime

In functional PLS, a variable stands for a fixed, but a possibly unknown, value. In imperative and object-oriented PLs, a variable contains a value. The variable may be inspected and updated as often as desired. Such a variable can be used to model a real-world object whose state changes over time.

To understand imperative variables, we assume an abstract storage model. A store is a collection of cells, each of which has a unique address. Each cell is either allocated or unallocated. Each allocated cell contains either a simple value or undefined. An allocated cell can be inspected, unless it contains undefined. An allocated cell can be updated at any time.

There are two classes of variables- simple and composite variables. A simple variable is one that contains a primitive value or a pointer. A simple variable occupies a single allocated cell. A composite variable is one that contains a composite value. A composite variable occupies a group of adjacent allocated cells.

A variable of a composite type has the same structure as a value of the same type. For instance, a tuple variable is a tuple of component variables. Similarly, an array variable is a mapping from an index range to a group of component variables. Depending on the PL, a composite variable can be totally updated (all at once), and/or selectively updated (one component at a time).

We illustrate declaration and updating of composite variables in C:

```
struct Date {int y, m, d;}
struct Date xmas, today;

// selective updating
xmas.d = 25;
xmas.m = 12;
xmas.y = 2022;

// total updating
today = xmas;
```

Every variable is created at some definite time, and destroyed at some later time when it is no longer needed. A variable's lifetime is the interval between its creation and destruction. A variable occupies cells only during its lifetime. When the variable is destroyed, these cells may be de-allocated. Moreover, these cells may subsequently be re-allocated to other variable(s).

A global variable's lifetime is the program's entire runtime. It is created by a global declaration. A local variable's time is an activation of a block. It is created by a declaration within that block, and destroyed on exit from that block. A heap variable's lifetime is arbitrary, but can be bounded by the program's runtime. It can be created at any time (by an allocator), and may be destroyed at any later time. It is accessed through a pointer.

The following is a C program, with different variable types.

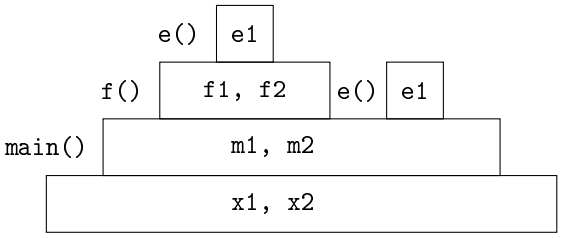
```
// global variables
extern int x1, x2;

void main() {
    // local variables
    int m1;
    float m2;
    // ..
    f();
    e();
}

void f() {
    // local variables
    float f1;
    int f2;
    // ..
    e();
}

void e() {
    // local variable
    int e1;
}
```

The following is the lifetime of global and local variables.



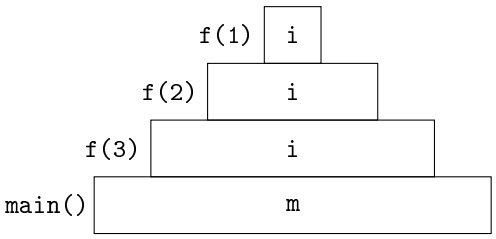
Global and local variable's lifetimes are nested. They cannot overlap.

Next, we consider a recursive program.

```
void main() {
    float m;
    // ..
    r(3);
}

void r(int n) {
    int i;
    if (n > 1) {
        // ..
        r(n-1);
    }
}
```

The following is the lifetime of the local variables.

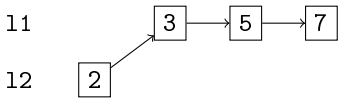


A local variable of a recursive procedure/function has several nested lifetimes.

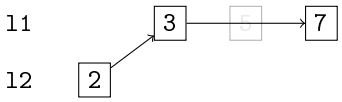
Now, we consider heap variables. Consider the C program below.

```
1  struct IntNode {int elem; IntList succ;}
2  typedef struct IntNode * IntList;
3
4  IntList c (int h, IntList t) {
5      IntList ns = (IntList) malloc (sizeof IntNode);
6      ns -> element = h;
7      ns -> succ = t;
8      return ns;
9  }
10
11 void d (IntList ns) {
12     ns -> succ = ns -> succc -> succ;
13 }
14
15 void main() {
16     IntList l1, l2;
17     l1 = c(3, c(5, c(7, NULL)));
18     l2 = c(2, l1);
19     d(l1);
20 }
```

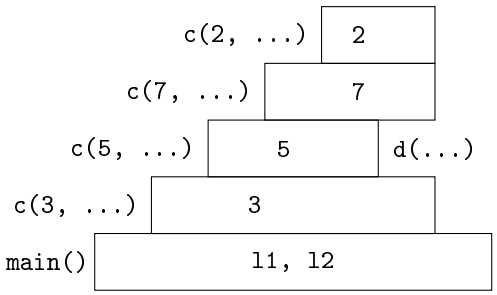
The variables `l1` and `l2` are heap variables since they are initialised using the `malloc` keyword. The following is the heap representation of `l1` and `l2`.



After we call `d(l1)`, the state of `l1` and `l2` is the following.



As we can see, the value `5` is not reachable after calling `d`. The lifetime of the local and heap variables is given below.



The lifetime of heap variables can overlap each other, and other local/global variable's lifetimes.

An allocator is an operation that creates a heap variable, yielding a pointer to the heap variable. For example, in C, the function `malloc` is the allocator, and in Java, an expression of the form `new C(..)` is an allocator. A deallocator is an operation that explicitly destroys a designated heap variable. For example, in C, the function `free` is the deallocator. Java does not have a deallocator.

A heap variable remains reachable as long as it can be accessed by following pointers from a local or a global variable. A heap variable's lifetime extends from its creation until:

- it is destroyed by a deallocator,
- it becomes unreachable, or
- the program terminates.

A pointer is a reference to a particular variable. A pointer's referent is the variable to which it refers. A null pointer is a special pointer value that has no referent. A pointer is essentially the address of its referent in the store. However, each pointer also has a type, and the type of a pointer allows us to infer the type of its referent.

Pointers and heap variables can be used to represent recursive variables such as lists and trees. But, the pointer itself is a low-level concept. Manipulation of pointers is notoriously error-prone and hard to understand. For example, the C assignment `p -> succ = q;` appears to manipulate a list, but which list? Also,

- Does it delete nodes from the list?
- Does it stitch together parts of two different lists?
- Does it introduce a cycle?

A dangling pointer is a pointer to a variable that has been destroyed. Dangling pointers arise when

- a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator
- a pointer to a local variable still exists at exit from the block in which the local variable was declared

A deallocator immediately destroys a heap variable. All existing pointers to that heap variable then become dangling pointers. For this reason, deallocators are inherently unsafe.

This means that C is somewhat unsafe and Java is safe. After a heap variable is destroyed in C, pointers to it might still exist. The same is true for local variables after we have exited their block. Java has no deallocator, so pointers to local variables cannot be obtained.

Now, consider the following C code.

```c
struct Date {int y, m, d;}
typedef Date * DatePtr;

// date1 points to a newly-allocated heap node
DatePtr date1 = (DatePtr) malloc (sizeof Date);
date1->y = 2022;
date1->m = 1;
date1->d = 1;

// date2 points to the same heap variable
DatePtr date2 = date1;

// deallocates that heap variable- date1 and date2 now contain
// dangling pointers
free(date2);

// behaves unpredictably
printf("%d4", date1->y);
date2->y = 2009;
```

A command (often called a statement) is a program construct that will be executed to update variables. Commands are characteristic of imperative and object-oriented PLs. Simple commands include:

- A skip command is a command that does nothing.
- An assignment command is a command that uses a value to update a variable.
- A procedure call is a command that calls a proper procedure with argument(s). Its net effect is to update some variables.

Compound commands include:

- A sequential command is a command that executes its sub-commands in sequence.
- A conditional command is a command that chooses one of its sub-commands to execute.
- An iterative command is a command that executes it sub-command repeatedly. This may be definite iteration, where the number of repetitions is known in advance, or an indefinite iteration, where the number is not known.
- A block command is a command that contains declarations of local variables.

Below are some examples in Java.

```
// single assignment
m = n + 1;

// multiple assignment
m = n = 0;

// assignment with a binary operator
m += 7;
n /= b;

// if command
if (x > y) {
    System.out.println(x);
} else {
    System.out.println(y);
}

// switch command
Date today = ...;
switch (today.m) {
    case 1:
        System.out.println("JAN");
        break;
    ...
    case 12:
        System.out.println("DEC");
        break;
}

// while command
Date[] dates;
// ...
int i = 0;
while (i < dates.length) {
    System.out.println(dates[i]);
    i++;
}

// for command
for (int i = 0; i < dates.length; i++) {
    System.out.println(dates[i]);
}

// block command
if (x > y) {
    int z = x;
    x = y;
    y = z;
}
```

The primary purpose of evaluating an expression is to yield a value. In most imperative and object-oriented PLs, evaluating expressions can also update variables. These are side effects. In C and Java, the body of a function is a command. If that command updates a global or a heap variable, calling the function has side effects. In C and Java, assignments are expressions with side effects- `V = E` stores the value of `E` in `V`, as well as yielding that value.

The C function `getchar(fp)` reads a character and updates the file variable `fp` points to. So, the following C code is correct.

```
char ch;
while ((ch = getchar(fp)) != NULL) {
```

```
    putchar(ch);
}
```

On the other hand, the following C code is incorrect.

```
enum Gender {FEMALE, MALE};
Gender g;
if (getchar(fp) == "F") {
    g = FEMALE;
} else if (getchar(fp) = "M") {
    g = MALE;
}
```

This is wrong since the second invocation of `getchar(fp)` has moved past the character.

3.2 Bindings and Scope

The meaning of an expression/command depends on the declarations of any identifiers used. A binding is a fixed association between an identifier and an entity (such as a value, variable or procedure). An environment (or namespace) is a set of bindings.

Each declaration produces some bindings which are added to the surrounding environment. Each expression/command is interpreted in a particular environment. Every identifier used in the expression/command have a binding in that environment.

The following is a C program outline, showing the environments for the two functions.

```
extern int z;
extern const float c = 3.0e6;

void f() {
    // ..

    // ENVIRONMENT:
        // c -> the FLOAT value 3 X 10^6
        // f -> a VOID -> VOID function
        // g -> a FLOAT -> VOID function
        // z -> an INT global variable
}

void g (float x) {
    char c;
    int i;
    // ..

    // ENVIRONMENT:
        // c -> a CHAR local variable
        // f -> a VOID -> VOID function
        // g -> a FLOAT -> VOID function
        // i -> an INT local variable
        // x -> a FLOAT local variable
        // z -> an INT global variable
}
```

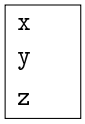
The scope of a declaration (or of a binding) is the portion of the program text over which it has effect. In some early PLs, the scope of every declaration was the whole program. In modern PLs, the scope of each declaration is controlled by the program's block structure.

A block is a program construct that delimits the scope of any declarations within it. Each PL has its own forms of blocks:

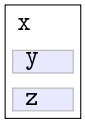
- In C, block commands (`{ ... }`), function bodies and compilation units are used;
- In Java, block commands (`{ ... }`), method bodies and class declarations are used;
- In Haskell, block expressions (`let ... in ...`), function bodies and modules are used.

A PL's block structure is the way in which blocks are arranged in the program text.

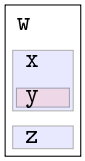
Some PLs such as Cobol have monolithic block structure- the whole program is a single block. The scope of every declaration is the whole program. This is depicted in the figure below.



Some PLs such as Fortran have flat block structure- the program is partitioned into blocks, but these blocks may not contain inner blocks. This is depicted in the figure below.



Modern PLs have nested block structure- blocks may be nested freely within other blocks. This is depicted in the figure below.



With nested block structure, the scope of a declaration excludes any inner block whether the same identifier is declared.

For example, C has a flat block structure for functions, but nested block structure for variables. Having a flat block structure for functions means that we cannot declare functions within functions, but that they can be accessed from anywhere within the program.

A binding occurrence of identifier I is an occurrence of I where I is bound to some entity e. An applied occurrence of identifier I is an occurrence of I where use is made of the entity e to which I is bound. If the PL is statically scoped, every applied occurrence of I should correspond to exactly one binding occurrence of I. Below we have a C program.

A PL is statically scoped if the body of a procedure is executed in the environment of the procedure definition. Then, we can decide at compile-time which binding occurrence of an identifier corresponds to a given applied occurrence.

A PL is dynamically scoped if the body of a procedure is executed in the environment of the procedure call site. Then, we cannot decide until runtime which occurrence of an identifier corresponds to a given applied occurrence, since the environment may vary from one call site to another.

The following is a program.

```
1 const int s = 2;
2
3 int f (int x) {
```

```
4       return x * s;
5   }
6
7   void g (int y) {
8       print (f (y));
9   }
10
11  void h (int z) {
12      const int s = 3;
13      print (f (z));
14  }
```

In a statically scoped language, the value of `s` at line `4` will always be the value of the global variable `2`. In a dynamically scoped language, the value of `s` depends on where we call it- at line `8`, we have `s = 2`, but at line `13`, we have `s = 3` because of the local variable defined at line `12`. So, there are multiple applied occurences for a single identifier in a dynamically scoped language.

Dynamic scoping fits badly with static typing. Nearly all PLs are statically scoped. Only a few PLs, such as Smalltalk and Lisp are dynamically scoped.

A declaration is a program construct that will be elaborated to produce binding(s). A declaration may also have side effects, such as creating a variable. A definition is a declaration whose only effect is to produce binding(s). It has no side effects.

Simple declarations include:

- A type declaration binds an identifier to an existing or new type.
- A constant definition binds an identifier to a value, possible after computation.
- A variable declaration binds an identifier to a newly-created variable.
- A procedure definition binds an identifier to a procedure.

A recursive definition is one that uses the bindings it produces itself. In almost all PLs, recursion is restricted to type (or class) declarations and procedure (or method) declarations.

An example below is given of a recursive Java class, with a recursive method.

```
1   class IntList {
2       int head;
3       IntList tail;
4
5       static int length(IntList list) {
6           if (list == null) {
7               return 0;
8           } else {
9               return 1 + length(list.tail);
10          }
11      }
12  }
```

C struct type declarations may be recursive, but only via pointers. Also, C function definitions may be recursive. Below is the C version of the class `IntList` above.

```
1   struct IntList {
2       int head;
3       struct IntList * tail;
```

```
}

int length(IntList * list) {
    if (list == NULL) {
        return 0;
    } else {
        return 1 + length(list->tail);
    }
}
```

3.3 Abstraction

In programming, abstraction means the distinction between what a program unit does and how it performs the task. This supports a separation of concerns between the implementor (i.e. who codes the program unit) and the application programmer (i.e. who uses it). Program units include procedures, packages, classes and generic packages.

Procedural Abstraction

A proper procedure (or just procedure) embodies a command to be executed. A procedure call is a command. It causes the procedure's body to be executed. It results in the update of some variables.

A function procedure (or just function) embodies an expression to be evaluated. A function call is an expression. It causes the function's body to be evaluated. It results in a value (which is returned by the function).

Imperative PLs usually support both procedures and functions. For example, in Pascal and Ada, procedures and functions are syntatically distinct. However, in C and Java, the only distinction is that a procedure's result type is `VOID`. Functional PLs only support functions. Object-oriented PLs also support procedures as methods. A static method is a procedure exported by a class. An instance method is a procedure attached to an object.

The following is a procedure in C.

```
void print(Date date) {
    int y = date.y,
        m = date.m,
        d = date.d;
    printf("%d4-%d2-%d2", y, m, d);
}
```

The procedure body is a block command. The execution of the procedure results in the variables `y`, `m` and `d` being created.

The following is a function in Haskell.

```
power   ::  (Float, Int) -> Float
power (b, n) =
    if n == 0
    then 1.0
    else b * power(b, n-1)
```

The body of the function is an expression. The following is the same function in C.

```
float power(float b, float n) {
    float p = 1;
    while (n > 0) {
        p *= b;
        n--;
    }
    return p;
}
```

Here, the body of the function is still a block command.

In most imperative and object-oriented PLs, the function's body is syntatically a block command. This is executed until a return determines the function's result. This means that the full expressive power is available to define

a function (e.g. instantiate variables). However, this is a roundabout way to compute a result. In fact, a return might never be executed (e.g. because of an infinite loop). Moreover, side effects are possible.

In functional PLs, the function's body is syntatically an expression. This is evaluated to yield the function's result. This means that the design is simple and natural. However, the expressive power of the PL is limited. This can be remedied by allowing conditional and iterative expressions, nonetheless.

An argument is a value (or another entity) that is passed to a procedure. An actual parameter is an expression that yields an argument. A formal parameter is an identifier through which a procedure can access an argument. We can pass the following as arguments:

- values,
- variables, or pointers to variables, and
- procedures, or pointers to procedures.

A parameter mechanism is a means by which a formal parameter provides access to the corresponding argument. Different PLs support a variety of parameter mechanisms: value, result, value-result, constant, variable, procedural and functional parameters. These can all be understood in terms of two underlying concepts: copy parameter mechanisms and reference parameter mechanisms.

With a copy parameter mechanism, a value is copied into and/or out of a procedure. A formal parameter is bound to a local variable of the procedure. A value is copied into that local variable on calling the procedure. A value is copied into that local variable on calling the procedure. The value can also be copied out of that local variable to an argument variable on return. There are 2 copy parameter mechanisms- copy in and copy out parameter.

In copy-in parameter (or value parameter), the argument is a value. On call, a local variable is created and initialised with the argument value. On return, the local variable is destroyed.

In copy-out parameter (or result parameter), the argument is a variable. On call, a local variable is created but not initialised. On return, the local variable's final value is assigned to the argument variable, then the local variable is destroyed.

The following is some functions in C.

```
1 void print(Date date) {
2     printf("%d-%d-%d", date.y, date.m, date.d);
3 }
4
5 void main() {
6     Date today = {2022, 3, 7};
7     print(today);
8 }
```

At line 1, the local variable `date` is initialised to the argument value. At line 7, the argument value is `{2022, 3, 7}`.

With a reference parameter mechanism, the formal parameter is a reference to the argument. The formal parameter FP is bound to a reference to the argument. Every accessed to FP is an indirect access to the argument. There are

3 principal reference parameter mechanisms- constant, variable and procedural parameter.

In a constant parameter mechanism, the argument is a value. So, any inspection of the FP is an indirect inspection of the argument value. In a variable parameter mechanism, the argument is a variable. So, any access (inspection or update) of the FP is an indirect access of the argument variable. In a procedural parameter mechanism, the argument is a procedure. So, any call to the FP is an indirect call to the argument procedure.

The following is some methods in Java.

```
1 public void print(Date date) {
2     System.out.println(date.y & "-" date.m & "-" & data.d);
3 }
4
5 public static void main() {
6     Date today = new Date(2022, 3, 7);
7     print(today);
8 }
```

At line 1, the local variable `date` is a reference to the argument object. At line 7, the argument is the object to which `today` refers. Java supports copy-in parameter mechanism for primitive types. Also, Java supports the reference parameter mechanism for object types.

Data abstraction

A package (or module) is a named group of components declared for a common purpose. These components may be types, constants, variables, procedures, inner packages, etc. The meaning of a package is the set of bindings exported by the package. These are often called the packages's application program interface (API).

An example is given below in Python.

```
1 words = [...]
2
3 def contains(word):
4     global words
5     return word in words
6
7 def add(word):
8     global words
9     if word not in words:
10        words += [word]
```

This module's API contains the variable `words` and the functions `contains` and `add`.

Some of the components of a program unit (package/class) may be private. This is called encapsulation. There are many levels of privacy, as given below.

- A component is private if it is visible only inside the program unit.
- A component is protected if it is visible only inside the program unit and certain other program units.
- A component is public if it is visible to application code outside the program unit.

A program unit API consists of its public bindings only.

Most PLs (such as Ada, java, Haskell) allow individual components of a program unit to be specified as private, protected or public. Python has a convention that components whose names start with "_" are private, and anything else is public. This is not enforced by the Python compiler.

In Java, the components of a package are classes and inner packages. Package components are added incrementally. The following is the outline of a class declaration within a package.

```
1 package sprockets;
2
3 import widgets.*;
4
5 public class C {
6     ...
7 }
```

At line 1, we declare that the class `C` is a component of the package `sprockets`. At line 3, we declare that the class `C` uses public components of the pakcage `widgets`.

An object is a tagged tuple of variable components (instance variables), equipped with operations that access these instance variables. A constructor is an operation that initialises a newly created object. An instance method is an operation that inspects and/or updates an existing object of the class. That object (called the receiver object) is determined by the method call. A class is a set of similar objects. All objects of a given class have similar instance variables, and are equipped with the same operations.

A Java class declaration declares its instance variables and defines its constructors and instance methods. It also specifies whether each of these is private, protected or public. A Java instance method call has the form `O.M(...)`. The expression `O` yields the receiver object. `M` is the name of the instance method to be called. The call executes the method body, with `this` bound to the receiver object `O`.

We illustrate this with an example.

```
1  class Dict {
2      private int size;
3      private String[] words;
4
5      public Dict(int capacity) {
6          ...
7      }
8
9      public void add (String w) {
10         if (!this.contains(w)) {
11             this.words[this.size++] = w;
12         }
13     }
14
15     public boolean contains (String w) {
16         ...
17     }
18 }
```

A possible application code is the following.

```
1 Dict mainDict = new Dict(10000);
2 Dict userDict = new Dict(100);
```

```
...
if (!mainDict.contains(currentWord) && !userDict.contains(
    currentWord)) {
    userDict.add(currentWord);
}
```

It is not however possible to do the following.

```
userDict.size = 0;
System.out.println(userDict.words[0]);
```

This is because the two instance variables are private.

If C' is a subclass of C (or C is a superclass of C'), then C' is a set of objects that are similar to one another, but richer than the objects of class C. In particular, an object of class C' has all the instance variables of an object of class C. But, it can have extra instance variables. Similarly, an object of class C' is equipped with all the instance methods of class C. But, it can also override some of them, and there might be extra instance methods.

By default, a subclass inherits (or shares) it superclass' instance methods. Alternatively, a subclass may override some of its superclass' instance methods by providing more specialised versions of these methods.

To illustrate this, consider the following class in Java.

```
class Shape {
    protected float x, y;

    public Shape() {
        this.x = 0.0;
        this.y = 0.0;
    }

    public final void move(float dx, float dy) {
        this.x += dx;
        this.y += dy;
    }

    // draw a point at (x, y)
    public void draw() {
        ...
    }
}
```

We define the subclass `Circle`.

```
class Circle extends Shape {
    private float r;

    public Circle(float radius) {
        this.x = 0;
        this.y = 0;
        this.r = radius;
    }

    // draw a circle at (x, y) of radius r
    @override
    public void draw() {
        ...
    }

    public float diameter() {
        return 2.0 * r;
    }
```

```
19 }
```

A possible application code is given below.

```
1 Shape s = new Shape();
2 Circle c = new Circle(10.0);
3 s.move(12.0, 5.0);
4 c.move(3.0, 4.0);
5
6 System.out.println(c.diameter());
7 s.draw();
8 c.draw();
9
10 s = c;
11 s.draw();
```

At line 7, we draw a point at `(12, 5)`. At line 8, we draw a circle at `(3, 4)` with radius `10`. We draw the same circle at line 11. This is dynamic dispatch- we draw a circle since `s` is a `Circle`.

Each instance method of a class C is inherited by the subclass C', unless it is overriden by C'. The overriding method in class C' has the same name and type as the original method in class C. Most object-oriented PLs allow the programmer to specify whether an instance method is `virtual` (may be overriden) or not. For example, in C++, an instance method specified by `virtual` may be overriden. In Java, an instance method specified by `final` cannot be overriden.

In every object-oriented PL, a variable of type C may refer to an object of any subclass of C. If a method M is virtual, then the method call `O.M` entails dynamic dispatch. The compiler infers the type of O as class C. It then checks that class C is equipped with an instance method named M of the appropriate type. At run-time, however, it might turn out that the receiver object is of class C', a subclass of C. The receiver object's tag is used to determine its actual class, and hence determine which of the methods named M is to be called.

An object-oriented PL supports single inheritance if each class has at most one superclass. Single inheritance gives rise to a hierarchy of class. Single inheritance is supported by most object-oriented PLs.

Multiple inheritance allows each class to have any number of superclasses. It is supported by C++. Multiple inheritance gives rise to both conceptual and implementation problems. For example, if both the superclasses override a method, then which version should be used within the subclass. To mitigate this, we might require the subclass to override the method, or use a priority system with the superclasses.

Generic abstraction

A program unit is generic if it is parameterised with respect to a type on which it depends. Many reusable program units (e.g. stack, list) are naturally generic. Generic program units include: generic packages, generic classes and generic procedures.

In Java, a generic class GC is parameterised with respect to a type T on which it depends.

```
1 class GC<T> {
2     ...
```

```
}
```

The generic class must be instantiated by substituting a type argument A for the type parameter T, i.e. `GC<A>`. This instantiation generates an ordinary class.

The following is a declaration of a generic class.

```
class List<T> {
    private static final cap = 100;
    private int size;
    private T[] elems;

    public List() {
        size = 0;
        elems = (T[]) new Object[cap];
    }

    public void add(T elem) {
        elems[size++] = elem;
    }
}
```

We can then make use of this class as follows.

```
public static void main() {
    List<String> sentence;
    sentence = new List<String>();
    sentence.add("...");
}
```

In Java, the type argument must be a class, not a primitive type.

Java also supports generic interfaces, such as `Comparable`.

```
interface Comparable<T> {
    public int compareTo(T that);
}
```

If a class C is declared as implementing `Comparable<C>`, then C must be equipped with a `compareTo` method that compares objects of type C.

Consider a generic class `GC<T>` that requires `T` to be equipped with some specific methods. Then, `T` may be specified as bounded by a class `C`.

```
class GC<T extends C> {
    ...
}
```

`T` is known to be equipped with all the methods of `C`, so we can make use of them within this class. Now, the type argument must be a subclass of `C`. Alternatively, `T` may be specified as bounded by an interface `I`. Moreover, `GC<T>` actually stands for `GC<T extends Object>`. The type argument must be a subclass of `Object`, which applies to all the classes.