CPS323 Lecture: Names, Binding, Lifetime, Scope 1/13/09

Objectives:

1. To introduce the concept of binding
2. To discuss alternatives for name binding and object lifetime
3. To discuss lexical and dynamic scope, and the related concept of referencing environments
4. To discuss and distinguish the concepts of aliasing, overloading, and polymorphism

Materials:

1. Pascal redefinition of true example to project/demonstrate
2. Projectable of Figure 3.10 in book
2. Projectable of Figure 3.4 in book

I. Names
- -----

A. As you have seen/will see in CPS311, at the machine language level, everything is represeted by numbers. But all programming languages (assembly as well as higher level) use symbolic names for a variety of purposes - e.g. variables, constants, executable code (methods, procedures, subroutines, functions, even whole programs), data types, classes, etc.

B. In general, names are of two different types:

1. Special symbols: +, -, * ...

2. Identifiers: sequences of alphanumeric characters (in most cases beginning with a letter), plus in many cases a few special characters such as '_' or '$'.

a. Programming languages are either case sensitive or case insensitive. In the former case, a and A are regarded as two distinct identifiers; in the latter case, a and A are the same. (Some languages only allow one case - e.g. until F90 only uppercase letters were allowed for identifiers in FORTRAN).

b. Most programming languages impose some limit on the number of characters in an identifier - examples

ASK

c. Many programming languages have conventions regarding identifiers which, while not enforced by the compiler, are generally observed by "native speakers".

i. Example: mixed case and lowercase/uppercase first letter conventions of Java

ii. Example: COBOL allowed hyphen (-) in identifiers, and the convention was to use this to separate words - e.g.

employee-name
print-report

iii. Example: Ada is case-insensitive, but the Ada83 reference manual used the convention of lowercase for reserved words and uppercase for identifiers, with underscores used to separate words - e.g.

```
if HOURS_WORKED > 40 then
    GROSS_PAY = GROSS_PAY + 0.5 * RATE * (HOURS_WORKED - 40);
end if
```

iv. In your projects, you should try to follow the conventions of whatever langauge you are using.

C. Most programming languages have the concept of either reserved words or keywords.

1. A reserved word is an identifier that has a special meaning in the language, and cannot be used for any other purpose (e.g. "while" in Java)

2. A keyword is an identifier that has a special meaning in the language, but can be used as a name in other contexts.

   Example: The following is legal FORTRAN

   ```
   IF (IF.EQ.3) IF = 4
   ```

3. Advantages of each?

   ASK

   a. An advantage of the reserved word approach is that programs are less confusing (though arguably something like the FORTRAN example I just gave represents really poor practice even if it is legal)

   b. An advantage of the keyword approach is that the introduction of a new reserved word in a language update does not "break" old code.

      Example: assert in Java.

      (It should be noted, though, that the designers of Java did intend anticipate problems like this by including in their reserved word list some words that Java does not use (like cast) but might use in the future. However, this was one they missed)

D. Some programming languages have the concept of a predeclared identifier.

1. A predeclared identifier is a name that is "built-in" to the language, but which the programmer can change. (This differs from a keyword, in that a redefinition totally changes the meaning. In contrast, in the FORTRAN keyword example, IF actually meant two different things in the same statement depending on context.)

2. Example: Consider the following Pascal fragment. What will it do?

```
program MessedUp(input, output);

const true = false;
begin
   if true = false then writeln('Postmodernism reigns')
end.
```

Display, compile on laptop (gpc), run

3. A predeclared identifier behaves as if there were a declaration for it in an outer scope that surrounds the entire program.

II. Binding/Binding Time

A. Definition:

1. A binding is an association between two things.

2. Binding time is the time when an association is established.

B. The concepts of binding and binding time are pervasive ones in computer science.

1. We consider these concepts here in the context of names in programming languages - e.g. the binding of a class name to a class or a variables name to a variable.

2. Other illustrative examples:

   a. Somthing other than a name being bound in a programming language: the binding of a data type to a variable in a programming language.

   b. A name being bound outside the context of a programming language: the binding of a name to a network socket (for which Unix actually provides a routine named bind).

C. In the context of programming languages, there are quite a few alternatives for binding time.

1. Language design time

   Example: reserved words are bound to their meanings at the time a given language is designed/redesigned

2. Language Implementation Time

   Example: the C language does not specify the range of values for the type int. Implemntations on early microcomputers typically used 16 bits for ints, yielding a range of values from -32768 to +32767. On early large computers, and on all computers today, C implementations typically use 32 bits for int.

   OTOH: in Java, the use of 32 bits for int is defined as part of the language.

3. Program write time

   Example: many names are bound to specific meanings when a person writes a program

4. Compile time: the time when a single compilation unit is compiled

   Example: addresses in program code inside the unit

5. Link time: the time when all the compilation units comprising a single program are linked as the final step in building the program. (A step you may not even be conscious of)

   Example: in most languages (though not Java), addresses of routines in other compilations units

6. Load time: the time when an executable program is loaded into memory in order to execute it

   Example: Actual physical memory addresses of static variables and code is often bound when the program is loaded.

7. Run time: while the program is actually running.

   Example: Actual phyical memory addresses of variables created by "new" is bound when the "new" operation is executed

D. Oftentimes, these are simplified into two possiblities.

1. "Dynamic binding" is used to refer to anything bound while the program is running.

2. "Static binding" is used to refer to anything bound prior to that time - e.g. from Language Design time through Load Time.

3. Examples: in a strongly-typed language such as Java, though the values of variables are bound dynamically, the types of variables are bound statically. In contrast, there are dynamically-typed languages, where even the types of variables are bound at run time.

III. Lifetime

A. The word "lifetime" is used in two slightly different ways

1. To refer to the lifetime of an _object_. (Here, we intend the term in a more general sense than the way it is used in OO: we mean any entity that exists while a program is running, whether the program is explicitly OO or not.)

2. The lifetime of the binding of a name to an object.

3. In the simplest case, the two are equivalent - but that is not always so.

a. An object can exist before the binding of a particular name to it

   Example (Java)

```
void something(Object o) {
    // 2
}
....
Object p = new Object()
// 1
....
something(p);
....
// 3
```

   The object named by p exists at point 1, but the name o is not bound to it until point 2

b. An object can exist after the binding of a particular name to it has ceased

   Example (the above)

   The object named by p continues to exist at point 3, even though the binding of the name o to it ended when method something() completed

c. A name can be bound to an object that does not yet exist

   Example (Java)

```
Object o;
// 1
....
o = new Object(); // 2
```

   At point 1, the name o is bound to an object that does not come into existence until point 2

d. A name can be bound to an object that has actually ceased to exist

   Example (C++ - not possible in Java)

```
Object o = new Object();
...
delete o;    // 1
...
// 2
```

   At 2, the name o is bound to an object that has ceased to exist. (The technical name for this is a dangling reference).

e. It is also possible for an object to exist without having any name at all bound to it.

Example (Java or C++, with slightly different nuances to be discussed later)

```
Object o = new Object();
// 1
...
o = new Object();
// 2
```

At point 2, the object that o was bound to at point 1 still exists, but o no longer is bound to it. In the absence of any other name bindings to it between points 1 and 2, this object now has no name referring to it. (The technical name for this is garbage).

4. The remainder of our discussion will focus on _object_ lifetime

B. There are three general types of object lifetime

1. Static (permanent) lifetime. The object exists during the entire time the program is running.

   a. Global variables (though if the variable is a pointer/reference, the object it refers to may not exist)

      - The precise way of declaring global variables differs from language to language

        - In some languages (e.g. BASIC) any variable

        - In FORTRAN any variable declared in the main program or in a COMMON block

        - In Java, any class field explicitly declared static

        - In C/C++, any variable declared outside of a class, or (C++) declared static inside a class

   b. Static local variables - only available in some languages

      - FORTRAN: In some implementations (but not all) all variables (other than COMMON) declared in a subprogram

      - C/C++ local variables explicitly declared static

      ```
      int foo() {
         static int i;
         ...
      ```

      A static local variable retains its value from one call of a routine to another

   c. Many constants (but not constants that are actually read-only variables)

2. Stack (dynamic, automatic)

   a. Typically, the local variables and parameters of a method have stack lifetime. This name comes from the normal way of implementing routines, regardless of language. (Different languages may call them methods or functions or procedures or subroutines, but the typical implementation is the same.)

      - Since routines obey a LIFO call return discipline, they can be managed by using a stack composed of stack frames - one for each currently active routine.

        Example:

```
void d() { /* 1 */ }
void c() { ... d() ... }
void b() { ... c() ... }
void a() { ... b() ... }
int main() { ... a() ... }
```

        Stack at point 1:

```
-------------------
|   Frame for d   |   <-- top of stack
-------------------
|   Frame for c   |
-------------------
|   Frame for b   |
-------------------
|   Frame for a   |
-------------------
| Frame for main  |
-------------------
```

      - This approach can easily accomodate recursion as well - Example:

```
int fact(int n) {
    if (n == 0) return n;
    else return n * fact(n-1);
}
```

        Stack for a call to fact(3):

```
---------------------
| Frame for fact(0) |   <-- top of stack
---------------------
| Frame for fact(1) |
---------------------
| Frame for fact(2) |
---------------------
| Frame for fact(3) |
---------------------
```

   b. Typically, the parameters and local variables of a routine are stored in the routine's stack frame. A variable's address in memory is then represented as some offset relative to the base address of the stack frame for the routine in belongs to.

c. Stack lifetime is typically used, then, for local variables and parameters of a routine - with a couple of exceptions:

   - Local ariables explicitly declared static (if the language allows this)

   - Local ariables in a language that forbids recursion may us permanent lifetime - which means that their value is preserved from one call of a routine to another.

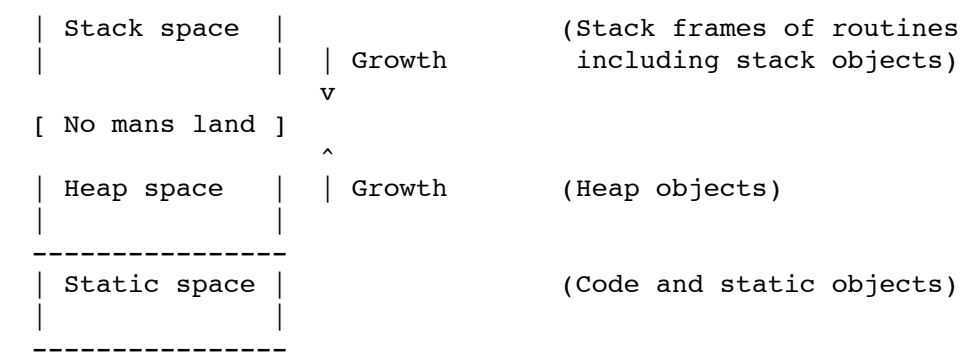
     At one time, for example, this was always the case with FORTRAN. Today, though, many FORTRAN implementations use stack lifetime for local variables, even though this is not required unless a routine is explicitly declared RECURSIVE (an F90 extension). [F77 added an explicit SAVE statement to force the older value-preserving behavior in an implementation that uses a stack]

3. Heap (Indefinite) lifetime is used for objects whose lifetime may only be a portion of the time a program is running, but does not obey a LIFO discipline.

   a. Objects allocated by new / deallocated by delete.

   b. Objects whose size can vary as the program is running - e.g. Strings, Collections.

   c. Typically, in a running program uses three different kinds of memory space, which may look like this:



   As the program is running, stack space can grow as routines are called, and shrink as they return. Heap space grows as objects are created. However, to prevent growth without limit, there must be some mechanism for recycling the storage used by objects that are no longer alive.

   d. Management of heap space presents some interesting challenges

      - A language implementation typically uses one of three approaches to "recycling" space used by objects that are no longer alive:

        - Explicit - the program is responsible for releasing space needed by objects that are no longer needed using some construct such as delete (C++)

- Problems:

  ASK

  - Dangling references

  - Storage leaks

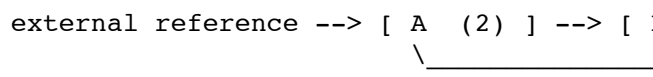
- Reference-counting: each heap object maintains a count of the number of external pointers/references to it. When this count drops to zero, the space utilized by the object can be reallocated

  - Problems:

    ASK

    - tricky to keep this information current

    - circular structures break this - e.g.



    if the external reference is dropped, A's reference count drops to 1, so it looks like it is still alive, even though both A and B should be subject to recycling!

- Garbage collection: Periodically, all existing objects are checked to see if there is any object with no outstanding references - in which case, it is garbage and can be recycled

  - This is the most elegant approach, but involves a fair amount of overhead

  - LISP, Java, and C#, among others, use this

- A second issue that arises if objects have differing sizes (as they typically do), is Compaction of storage to avoid fragmentation

IV. Scope

A. Definition: the scope of a name binding is the portion of the text of the (source) program in which that binding is in effect - i.e. the name can be used to refer to the corresponding object. Using a name outside the scope of a particular binding implies one of two things: either it is undefined, or it refers to a different binding.

Example:

```
class Foo
{
  private int n;

  void foo() {
      // 1
  }

  void bar() {
      int m,n;
      ...
      // 2
  }
  ...
```

A reference to m at point 1 is undefined
A reference to n at point 1 refers to an instance variable of Foo; at point 2 it refers to a local variable of bar.

B. There are two general approaches that programming languages have taken to the scope issue.

1. Dyanamic scope (early LISP, but a non-default option ion Common LISP and otherwise very rare). The scope of a binding is determined as the program is running; a name always refers to the most recent binding for that name.

2. Lexical (static) scope: The scope of a binding is determined by the structure of the program's text. What a name refers to can always be determined at compile time.

3. The book gives an example of the difference between these two, using a (hypothetical) implementation of Pascal which allows a choice of which sort of scope to use. (Pascal is actually always statically scoped)

   PROJECT, DISCUSS Figure 3.10 from the text

   In both cases, there is a variable a in the main program that is initialized to 2 in line 7, and written out in line 12

   - Dynamic scope: If the user enters a positive number, second is called and it in turn calls first. In line 3, a refers to the most recent binding for a at line 5 in second, so the value of the global variable a is unchanged, and 2 is written. If the user enters a negative or zero number, first is called directly, and the value of a is changed to 1, so 1 is written

   - Static scope: line 3 always refers to the variable a declared at line 1, so the program always writes 1

C. Since lexical scope is almost the universal convention in modern languages, we will discuss only this alternative. There are a number of subordinate issues.

1. Nested scopes. Many programming languages allow scopes to be nested inside each other.

   a. Example: The Java example defined earlier

   b. Example: Java actually allows classes to be defined inside classes or even inside methods, which permits multiple scopes to be nested. An extreme (though legal) example

```
class Outer
{
   int v1;  // 1

   void methodO()
   {
       float v2; // 2

       class Middle
       {
           char v3; // 3

           void methodM()
           {
               boolean v4; // 4

               class Inner
               {
                   double v5; // 5

                   void methodI()
                   {
                       String v6; // 6
                   }
               }
           }
       }
   }
}
```

The scope of the binding for v1 the whole program
The scope of the binding for v2 is methodO and all of classes Middle and Inner, including their methods
The scope of the binding for v3 is all of classes Middle and Inner, including their methods
The scope of the binding for v4 is methodM and all of class Inner, including its method
The scope of the binding for v5 is all of class Inner, including its method
The scope of the binding for v6 is just methodI

   c. Example: some programming languages - including Pascal and its descendants (e.g. Ada) - allow procedures to be nested inside procedures. (C and its descendants do not allow this)

   Example: PROJECT Figure 3.4

d. With nested scopes, one can have the phenomon of _hole in scope_ if a name is bound inside the scope of another binding for the same name

   Example: Suppose, in the above, that instead of binding v5 at point 5, we bound v1. Then the scope of the outer declaration of v1 (at point 1) would encompass all of classes Outer and Middle, while all references to v1 in class Inner would refer to the inner binding at point 5.

   When hole in scope occurs, the meaning of a name is determined by the nearest binding that contains the use of the name.

e. How is nested scope handled in terms of stack frames? Refer again to the earlier Pascal example

   PROJECT Figure 3.4 again

   - We say that P1 is a level 1 procedure; P2 and P4 are level 2; and P3 and F1 are level 3. (In general, the level of a routine is 1 more than that of the routine that directly contains it, with the outermost level being level 1).

   - Static scoping implies that a routine can call any routine at the same or a lower-numbered level, or a routine numbered one level higher, but then only if that routine is contained within it - e.g.

     P1 can only call P1, P2 or P4.
     P4 can only call P1, P2, P4 or F1
     F1 can only call P1, P2, P4 or F1 [since there is nothing inside it]
     ...
     P3 can only be called by P2
     F1 can only be called by P4

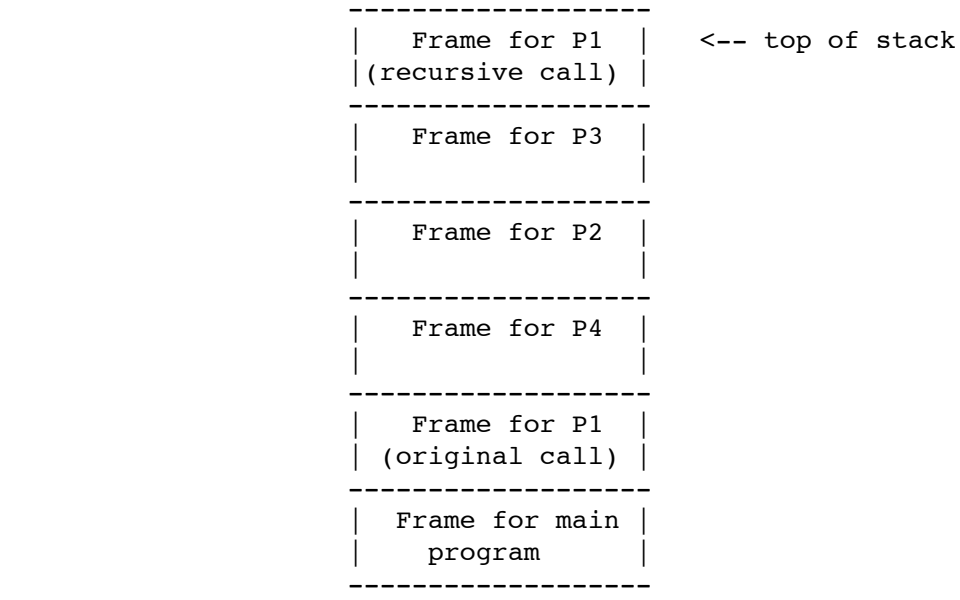
   - Static scoping also implies that a routine can only access its own variables, and those declared in routines that contain it - e.g.

     P1 can only access variables declared in P1
     P2 can only access variables declared in P2 or P1
     P3 can only access variables declared in P3, P2 or P1
     P4 can only access variables declared in P4 or P1
     F1 can only access variables declared in F1, P4, or P1

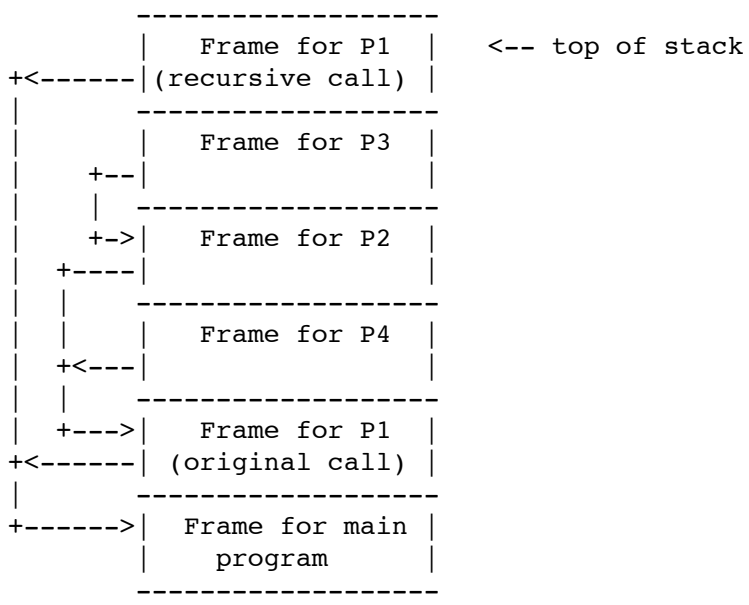
   - Now suppose the following occurs:

       P1 is called
       P1 calls P4
       P4 calls P2
       P2 calls P3
       P3 (recursively) calls P1

At this point, the stack looks like this



The rules for variable access can be implemented if each frame includes a link to the frame for what was the most recent call to the routine that immediately contains it at the time the routine was called. This is called the _static link_ - e.g.



Now, if a routine wants to access variables declared in a lower-numbered frame, it simply traverses as many links on the static chain as the level number differ - e.g.

P3 needs to access a variable in P1. The level numbers differ by 2, therefore we traverse two links on the static chain to get from the frame for P3 to the correct frame for P1 (the one that resulted in the call to P3)

2. Declaration order

   a. When a declaration appears other than at the very beginning of a block, does the scope of the declaration begin at the point of the declaration, or at the start of the block?

      Java exemplifies both of these at various points.

      - A field or method declared in a Java class can be used anywhere in the class, even before its declaration.

      - A local variable declared in a method cannot be used before the point of its declaration

      Example:

```
class Demo
{
    public void method()
    {
        // Point 1

        int y;
    }

    private int x;
}
```

      The instance variable x can be used at Point 1, but not y

   b. There are good reasons for both approaches

      - In favor of beginning scope at point of declaration: avoids need to read program twice.

      - In favor of beginning scope at start of block: required for mutually recursion.

        Example: Java

```
class Demo
{
    public void method1()
    {
        ...
        method2();
        ...
    }

    public void method2()
    {
        ...
        method1();
        ...
    }
}
```

        Languages that begin scope at point of declaration need to resort to other strategies to support mutual recursion: some sort of provision for "forward" or incomplete declarations.

Example: C/C++:

```
void method2(); // Incomplete declaration

void method1();
{
    ...
    method2();
    ...
}

void method2()  // Definition completes the abov
{
    ...
    method1();
    ...
}
```

c. A second question: if the scope of a name begins at the point of its declaration, and there is another declaration for the same name in an outer scope, what happens if the name is used before its declaration?

- The outer declaration is used
- The use is illegal

Example: Add private int y; as an instance variable in the above; now if x is used at point 1, does it refer to this declaration or is it illegal? (You will explore on homework)

3. The notion of elaboration at scope entry

a. We normally think of a declaration as simply giving meaning to a name, but sometimes a declaration also involves some computation.

Example: Robot karel = new Robot(...);

b. This computation is typically done whenever its scope is entered.

Example: in Java, if an instance field has an initializer, the initializer is executed whenever an instance of the class is created.

c. The phrase "elaboration of a declaration" is used by some languages (e.g. Ada) to refer to this. In some cases, a fair amount of computation may need to occur, depending on what is being declared.

D. Name spaces and qualified names

1. If a program is large, there is significant danger that the same name will be used for two different things - perhaps as a result of multiple programmers independently working on different parts of the program.

2. To prevent problems like this, many languages incorporate some sort of mechanism whereby a name is qualfied in some way.

Example: in Java, the name of a class is qualified by the name of the package in which it occurs - e.g. java.io.File

in Java, again, the name of a method or field is qualified by the name of a class or an object of that class - e.g. Math.PI or System.out.println.

Example: in C++, a name may be qualified by a namespace, separated from the name by :: - e.g. std::cout

3. Since always specifying qualifiers can generate a lot of typing, languages that have name spaces typically incorporate a facility for using a namespace or importing one or more names from a namespace

Example: in Java: import java.io.File; allows File to be used in the importing file without being qualified by java.io.

in Java: import static java.lang.Math.* allows _all_ static names defined in class Math to be used without being qualified by Math. - e.g. after the above, one could use PI instead of Math.PI.

in C++: using namespace std allows all names defined in namespace std to be used without qualification - e.g. after the above, one could use cout instead of std::cout

A similar facility exists in Ada.

V. Some Miscellaneous Issues

A. Aliasing

Two names are said to be aliases if they are names for the same thing.

1. Aliases can be created by assignment of pointers/references

Example: Java:

```
Robot karel = ...
Robot foo = karel;
```

foo and karel are aliases for the same Robot object

2. Aliases can be created by passing reference parameters

Example: C++

```
void something(int a [], int & b)
{
    // 1
    ...
}

int x [100];
int y;
something(x, y);
```

After the call to something(x, y), at point 1 x and a are aliases for the same array

3. In general, aliases make understanding a program (or proving its correctness) more challenging because changing an object through one alias affects its meaning when accessed through another alias as well.

B. Overloading

A name is said to be overloaded if, in some scope, it has two or more meanings, with the actual meaning being determined by how it is used.

Example: C++

```
void something(char x)
...
void something(double x)
...
// 1
```

At point 1, something can refer to one or the other of the two methods, depending on its parameter.

C. Overloading permits polymorphism.

1. Compile-time (static) polymorphism: the meaning of a name is determined at compile-time from the declared types of what it uses

   Example: in the above C++ example, something(3.14) will refer to the second definition of something; this determination is made at compile time

2. Run-time (dynamic) polymorphism: the meaning of a name is determined when the program is running from the actual types of what it uses.

3. Example: Java has both types in different contexts:

   a. When a name is overloaded in a single class, static polymorphism is used to determine which declaration is meant.

   b. When a name is overridden in a subclass, dynamic polymorphism is used to determine which version to use.

4. In contrast, C++ uses static polymorphism in both cases by default, though dynamic polymorphism can be used for overrides if a method is declared virtual.

D. Generics

Some languages permit generic (template) declarations, in which the definition of a name is parameterized.

Example: Java generics

Example: C++ templates