SIDEBARS LISKOV

- ## CAPITOLO 1: INTRODUCTION

Sidebar 1.1 Abstraction Mechanisms

- *Abstraction by parameterization* abstracts from the identity of the data by replacing them with parameters. It generalizes modules so that they can be used in more situations.
- *Abstraction by specification* abstracts from the implementation details (how the module is implemented) to the behavior users can depend on (what the module does). It isolates modules from one another's implementations; we require only that a module's implementation supports the behavior being relied on.

- Cosa significa in generale "**astrarre**" e in che termini viene usato all'interno di questo corso?
- Perché è bene nell'ambito della progettazione del software che si mettano in pratica determinati meccanismi di astrazione?
- Cosa si intende per **astrazione per parametrizzazione**? Fornire esempi a riguardo
- Cosa si intende per **astrazione per specificazione**? Fornire esempi a riguardo

Sidebar 1.2 Kinds of Abstractions

- *Procedural abstraction* allows us to introduce new operations.
- *Data abstraction* allows us to introduce new types of data objects.
- *Iteration abstraction* allows us to iterate over items in a collection without revealing details of how the items are obtained.
- *Type hierarchy* allows us to abstract from individual data types to families of related types.

- Saper introdurre e descrivere l'utilità e le caratteristiche di ciascuna delle tipologie principali di astrazione osservate durante il corso (astrazione procedurale, sui dati, iterazione)

- # CAPITOLO 2: UNDERSTANDING OBJECTS IN JAVA

Sidebar 2.2 Mutability and Sharing

- An object is *mutable* if its state can change. For example, arrays are mutable.
- An object is *immutable* if its state never changes. For example, strings are immutable.
- An object is *shared* by two variables if it can be accessed through either of them.
- If a mutable object is shared by two variables, modifications made through one of the variables will be visible when the object is used through the other.

- Definizione di oggetti **mutabili** ed **immutabili**, con relativi esempi
- Cosa significa dire che un oggetto è "**condiviso**" da più variabili? In quali circostanze lo sharing di un oggetto necessita una particolare attenzione? Perché? (Pensare ad esempi a riguardo)

Sidebar 2.3 Type Safety

- One important difference between Java and C and C++ is that Java provides type safety. This is accomplished by three mechanisms:
  - Java is a strongly typed language. This means that type errors such as using a pointer as an integer are detected by the compiler.
  - Java provides automatic storage management for all objects. In C and C++, programs manage storage for objects in the heap explicitly. Explicit management is a major source of errors such as *dangling references*, in which storage is deallocated while a program still refers to it.
  - Java checks all array accesses to ensure they are within bounds.
- These techniques ensure that type mismatches cannot occur at runtime. In this way an important source of errors is eliminated from your code.

- Cosa significa dire che Java è un linguaggio **fortemente tipizzato**?
- Cosa significa dire che Java fornisce la **type safety**? Tramite quali meccanismi essa viene garantita dal linguaggio?
- Cos'è **l'automatic storage management**? In cosa consiste?
- Gli errori sui tipi vengono intercettati solo in fase di compilazione? Quali tipi coinvolgono errori intercettati in fase di compilazione e quali invece riguardano errori intercettati a tempo di esecuzione?
- Cos'è e come funziona il meccanismo del **dispatching**? (Pensare ad esempi a riguardo)

Sidebar 2.4 Type Hierarchy

- Java supports *type hierarchy*, in which one type can be the *supertype* of other types, which are its *subtypes*. A subtype's objects have all the methods defined by the supertype.
- All object types are subtypes of `Object`, which is the top of the type hierarchy. `Object` defines a number of methods, including `equals` and `toString`. Every object is guaranteed to have these methods.
- The *apparent type* of a variable is the type understood by the compiler from information available in declarations. The *actual type* of an object is its real type—the type it receives when it is created.
- Java guarantees that the apparent type of any expression is a supertype of its actual type.

- Cosa significa dire che Java supporta una **gerarchia di tipi**?
- In base a quale relazione i tipi in Java sono disposti nella gerarchia? Qual è il tipo presente in cima a questa gerarchia? Quali metodi presenta? Perché è ragionevole che quel tipo abbia quei determinati metodi e che essi vengano ereditati da tutti i sottotipi?
- Qual è la differenza tra **tipi concreti** e **tipi apparenti**? La distinzione tra tipi apparenti e concreti può riferirsi a tutti e due i tipi messi a disposizione del linguaggio Java (tipi primitivi/riferimento)? Perché? (Pensare ad esempi a riguardo)
- Perché esistono i tipi apparenti e i tipi concreti? Quali sono le caratteristiche del linguaggio che ne giustificano l'esistenza?

- # CAPITOLO 3: PROCEDURAL ABSTRACTION

  - Cosa si intende per **astrazione procedurale**? Perché si utilizza? Quali sono i suoi benefici?

Sidebar 3.1 Benefits of Abstraction by Specification

- *Locality*—The implementation of an abstraction can be read or written without needing to examine the implementations of any other abstractions.
- *Modifiability*—An abstraction can be reimplemented without requiring changes to any abstractions that use it.

  - Cosa si intende per astrazione per specificazione? Quali sono i suoi benefici? Cosa garantisce?
  - Cosa significano i termini **località** e **modificabilità** associati a questo aspetto? Pensare a degli esempi
  - In che senso una specificazione può essere intesa su un **livello sintattico** e su un **livello semantico**?

Sidebar 3.2 Properties of Procedures and Their Implementations

- *Minimality*—One specification is more minimal than another if it contains fewer constraints on allowable behavior.
- *Underdetermined behavior*—A procedure is underdetermined if for certain inputs its specification allows more than one possible result.
- *Deterministic implementation*—An implementation of a procedure is deterministic if, for the same inputs, it always produces the same result. Implementations of underdetermined procedures are almost always deterministic.
- *Generality*—One specification is more general than another if it can handle a larger class of inputs.

  - Quali sono le proprietà più importanti di una specificazione? In altri termini, cosa sono **minimalità** e **generalità** e in che senso questi termini si applicano al contesto delle specificazioni?
  - Cosa significa dire che una procedura ha un **comportamento indeterminato**?
  - Cosa significa il termine "**deterministico**" applicato all'implementazione di una procedura?

Sidebar 3.3 Total versus Partial Procedures

- A procedure is *total* if its behavior is specified for all legal inputs; otherwise, it is *partial*. The specification of a partial procedure always contains a requires clause.
- Partial procedures are less safe than total ones. Therefore, they should be used only when the context of use is limited or when they enable a substantial benefit, such as better performance.
- When possible, the implementation should check the constraints in the requires clause and throw an exception if they are not satisfied.

- Cosa si intende per procedure **parziali** e **totali**? Quando è meglio utilizzare le une piuttosto che le altre?
- In che modo si rende totale una procedura parziale?
- E' sempre possibile rendere totale una procedura parziale?
- Quando non è ragionevole rendere una procedura totale, lasciando dunque che essa resti parziale? (Pensare ad esempi a riguardo)

- # CAPITOLO 4: EXCEPTIONS

  - In generale: cosa sono le **eccezioni**? A cosa servono/da che esigenza nascono? Quali potrebbero le altre possibilità per gestire dei comportamenti anomali all'interno del codice? Quali sono i difetti di queste altre possibilità che spingono dunque all'utilizzo del meccanismo delle eccezioni

Sidebar 4.1 Rules for Using Exceptions

- When the context of use is local, you need not use exceptions because you can easily verify that requires clauses are satisfied by calls and that special results are used properly.
- However, when the context of use is nonlocal, you should use exceptions instead of special results. And you should use exceptions instead of requires clauses unless a requirement cannot be checked or is very expensive to check.

  - In che contesto devono essere usate le eccezioni?
  - Che legame c'è tra l'utilizzo delle eccezioni e la parzialità/totalità delle procedure?

Sidebar 4.2 Checked versus Unchecked Exceptions

- You should use an unchecked exception only if you expect that users will usually write code that ensures the exception will not happen, because
  - There is a convenient and inexpensive way to avoid the exception.
  - The context of use is local.
- Otherwise, you should use a checked exception.

  - Qual è la differenza tra eccezioni **checked** e **unchecked**?
  - Quando usare le une, quando usare le altre?
  - Tutte le eccezioni devono essere gestite? Come si deve eventualmente gestire un'eccezione?
  - Cosa si intende per **graceful degradation** legata ad un'eccezione? (Pensare ad esempi)
  - Cosa prevede il meccanismo di masking/reflecting? Quando può essere opportuno usarlo per gestire delle eccezioni? (pensare a degli esempi)

- # CAPITOLO 5: DATA ABSTRACTION

  - In generale, cos'è **l'astrazione sui dati**? A cosa serve? Quali sono le sue motivazioni?
  - Qual è il costrutto del linguaggio che consente questo meccanismo?

Sidebar 5.1 `equals`, `clone`, **and** `toString`

- Two objects are `equals` if they are behaviorally equivalent. Mutable objects are `equals` only if they are the same object; such types can inherit `equals` from `Object`. Immutable objects are `equals` if they have the same state; immutable types must implement `equals` themselves.
- `clone` should return an object that has the same state as its object. Immutable types can inherit `clone` from `Object`, but mutable types must implement it themselves.
- `toString` should return a string showing the type and current state of its object. All types must implement `toString` themselves.

  - Qual è il contratto di **equals**? Ha senso parlare di uguaglianza nel senso stretto del termine se si ha a che fare con oggetti mutabili?
  - Quali sono gli altri metodi presenti in Object? Perché ha senso che in un contesto gerarchico come quello di Java tali metodi stiano nella classe Object?

Sidebar 5.2 Abstraction Function and Rep Invariant

- The abstraction function explains the interpretation of the rep. It maps the state of each legal representation object to the abstract object it is intended to represent. It is implemented by the `toString` method.
- The representation invariant defines all the common assumptions that underlie the implementations of a type's operations. It defines which representations are legal by mapping each representation object to either true (if its rep is legal) or false (if its rep is not legal). It is implemented by the `repOk` method.

  - Cos'è la **funzione di astrazione**? Qual è il suo dominio? Qual è il suo codominio? Cosa ci consente di fare? Qual è il metodo che fornisce un'implementazione alla funzione d'astrazione?
  - È più plausibile che una funzione di astrazione sia suriettiva o iniettiva? Perché? Pensare a casi in cui la funzione d'astrazione è molti-ad-uno e a casi in cui invece è una funzione uno-ad-uno
  - Cos'è **l'invariante di rappresentazione**? A cosa serve? Come viene implementato?

Sidebar 5.3 Properties of Data Abstraction Implementations

- An implementation performs a *benevolent side effect* if it modifies the rep without affecting the abstract state of its object. Benevolent side effects are possible only when the abstraction function is many-to-one.
- An implementation *exposes the rep* if it provides users of its objects with a way of accessing some mutable component of the rep.

- Cosa si intende per **benevolent side effect**? In che contesto se ne parla? Fare un esempio di un caso in cui compare un effetto collaterale benevolo
- Perché un effetto collaterale benevolo è possibile solo quando la funzione d'astrazione è molti ad uno?
- Un'astrazione mutabile implica una rappresentazione mutabile?
- Un'astrazione immutabile implica una rappresentazione immutabile?
- Cosa significa dire che un'implementazione **espone la sua rappresentazione**?
- Quali sono le criticità legate a questo aspetto? Fare degli esempi a riguardo

Sidebar 5.4 Reasoning about Data Abstractions

- Data type induction is used to reason about whether an implementation preserves the rep invariant. For each operation, we assume the rep invariant holds for any inputs of the type, and show it holds at return for any inputs of the type and any new objects of the type.
- To prove the correctness of an operation, we make use of the abstraction function to relate the abstract objects mentioned in its specification to the concrete objects that represent them.
- Data type induction is also used to reason about abstract invariants. However, in this case, the reasoning is based on the specification, and observers can be ignored.

- Cos'è **l'induzione sui tipi di dato**? A cosa serve? Come si concretizza?
- Dove è più critico (nonché necessario) controllare il preservamento dell'invariante di rappresentazione?
- Cosa si intende per correttezza delle operazioni? Cosa si utilizza per provare la correttezza delle operazioni? Perché?

Sidebar 5.5 Properties of Data Abstractions

- A data abstraction is mutable if it has any mutator methods; otherwise, the data abstraction is immutable.
- There are four kinds of operations provided by data abstractions: *creators* produce new objects "from scratch"; *producers* produces new objects given existing objects as arguments, *mutators* modify the state of their object; and *observers* provide information about the state of their object.
- A data type is *adequate* if it provides enough operations so that whatever users need to do with its objects can be done conveniently and with reasonable efficiency.

- Quando un'astrazione si definisce **mutabile**?
- Quali sono le categorie di metodi associate ad una generica astrazione? In altri termini, di che tipologie di metodi necessita (o potrebbe necessitare) un'astrazione per poter funzionare correttamente? (Cosa sono e a cosa servono i metodi di mutazione, quelli di osservazione e quelli di produzione? Sono sempre necessari metodi di tutte e tre le categorie per ogni tipologia di astrazione? Fare degli esempi a riguardo
- Cosa si intende per **adeguatezza** di un'astrazione? Quando un'astrazione è adeguata? Quando non lo è? Fare esempi legati a questo aspetto

Sidebar 5.6 Locality and Modifiability for Data Abstraction

- A data abstraction implementation provides locality if using code cannot modify components of the rep; that is, it must not expose the rep.
- A data abstraction implementation provides modifiability if, in addition, there is no way for using code to access any part of the rep.

- Cosa si intende se si parla di **località** e di **modificabilità** legata ad un'astrazione sui dati?
- In che senso un'implementazione fornisce località? In che senso fornisce modificabilità?

- # CAPITOLO 6: ITERATION ABSTRACTION

Sidebar 6.1 Iterators and Generators

- An *iterator* is a procedure that returns a *generator*. A data abstraction can have one or more iterator methods, and there can also be standalone iterators.
- A generator is an object that produces the elements used in the iteration. It has methods to get the next element and to determine whether there are any more elements. The generator's type is a subtype of `Iterator`.
- The specification of an iterator defines the behavior of the generator; a generator has no specification of its own. The iterator specification often includes a requires clause at the end constraining the code that uses the generator.

- In generale, cos'è il meccanismo di **astrazione per iterazione**? A che cosa serve? Da che esigenza nasce? Quali sono delle altre possibilità per rispondere a tale esigenza? Perché queste altre modalità non sono quelle di fatto utilizzate (che criticità coinvolgono)?
- Nella nomenclatura della Liskov, cosa si intende per **iterator** e cosa si intende per **generator**?
- Come può essere attuata a livello implementativo l'astrazione iterazione? In altri termini, in che modo in Java è possibile sfruttare il meccanismo di iterazione? Quali sono le caratteristiche del linguaggio che lo consentono?
- Come e dove deve avvenire la specificazione di un iteratore?
- Differenze tra **iterazione interna** ed **iterazione esterna**. Perché in Java si tende ad usare pochissimo quella interna, prediligendo dunque quella esterna?

Sidebar 6.2 Using Generators

- Using code interacts with a generator via the `Iterator` interface.
- Using code must obey the constraint imposed on it by the iterator's requires clause.
- Generators can be passed as arguments and returned as results.
- It is sometimes useful to *prime* the generator: to consume some of the produced items before looping over the rest of them.

- In che termini viene usata l'interfaccia **Iterator** nel contesto dell'iterazione?
- Seguendo invece un approccio più moderno di quello della Liskov, in che termini viene usata l'interfaccia **Iterable** nel contesto dell'iterazione? Quali sono le caratteristiche e le differenze nell'utilizzare queste due tipologie di approcci (inteso: nell'approccio in cui la classe in questione non implementa Iterable e in quello in cui invece Iterable viene implementato?

Sidebar 6.3 Implementing Iterators

- An iterator's implementation requires the implementation of a class for the associated generator.
- The generator class is a *static inner class*: it is nested inside the class containing the iterator and can access the private information of its containing class.
- The generator class defines a subtype of `Iterator`.
- The implementation of the generator assumes using code obeys constraints imposed on it by the requires clause of the iterator.

- Come vengono di fatto implementati gli iteratori all'interno delle nostre astrazioni?
- In questo contesto qual è l'utilità di avere una classe dentro un'altra classe (meccanismo delle **inner classes**)?
- Qual è la differenza sostanziale tra una inner class **statica** ed una **non statica**? Perché in ogni caso in contesti come quello dell'iterazione si preferisce usare una inner class rispetto ad una classe esterna rispetto a quella di riferimento?
- Cos'è una **classe anonima**? Come funziona e perché è comoda da utilizzare in questo contesto?

- # CAPITOLO 7: TYPE HIERARCHY

  - In generale, definire e presentare il meccanismo di **gerarchia dei tipi** proprio del linguaggio Java. A cosa serve? Cosa comporta questo meccanismo?

Sidebar 7.1 Type Hierarchy

- Type hierarchy is used to define type families consisting of a supertype and its subtypes. The hierarchy can extend through many levels.
- Some type families are used to provide multiple implementations of a type: the subtypes provide different implementations of their supertype.
- More generally, though, subtypes extend the behavior of their supertype, for example, by providing extra methods.
- The substitution principle provides abstraction by specification for type families by requiring that subtypes behave in accordance with the specification of their supertype.

  - Quali sono i motivi per cui in Java esiste una gerarchia di tipi? Per quali motivazioni siamo portati a definire una relazione gerarchica che coinvolge un tipo e un suo sottotipo?

Sidebar 7.2 Assignment and Dispatching

- The compiler deduces an *apparent* type for each object by using the information in variable and method declarations.
- Each object has an *actual* type that it receives when it is created: this is the type defined by the class that constructs it.
- The compiler ensures the apparent type it deduces for an object is always a supertype of the actual type of the object.
- The compiler determines what calls are legal based on the object's apparent type.
- Dispatching causes method calls to go to the object's actual code—that is, the code provided by its class.

  - In che modo le nozioni di tipo concreto e apparente si relazionano al concetto di gerarchia in Java?
  - Il riconoscimento di un tipo apparente è un processo che viene effettuato a tempo di compilazione o a tempo di esecuzione? Perché? Stessa domanda per i tipi concreti
  - Fare un esempio concreto in cui emerge la distinzione tra tipo concreto e tipo apparente, facendo anche rifermento al meccanismo di dispatching e al suo funzionamento

Sidebar 7.3 Defining a Hierarchy

- A supertype is defined by either a class or an interface, which provides its specification and, in the case of the class, provides a partial or complete implementation.
- An abstract class provides only a partial implementation; it has no objects and no constructors that users can call.
- A subclass can *inherit* the implementations of its superclass's methods, but it can also *override* those implementations (for nonfinal methods).
- The rep of a subclass consists of its own instance variables and those of its superclass, but it can access the superclass instance variables only if they are declared to be *protected*.

- Definire le differenze fondamentali tra **classi concrete**, **classi astratte** e **interfacce**. In che contesti è opportuno usare le prime, le seconde o le terze?
- Quale può essere il senso di avere un'interfaccia che ha per sottotipo un'altra interfaccia? Spiegare e fornire un esempio
- Cosa sono i metodi di default per le interfacce? A cosa possono servire?
- Il sig. Block afferma che in genere è preferibile utilizzare le interfacce al posto delle classi astratte. Perché? Nel caso di due classi l'ereditarietà è singola o multipla? E nel caso delle interfacce? Perché?
- Quando invece è preferibile e sensato utilizzare una classe astratta al posto di un'interfaccia?
- Cosa significa dire che una sottoclasse **eredita l'implementazione** dei metodi della superclasse? E cosa significa dire che nella sottoclasse si possono sovrascrivere dei metodi appartenenti alla superclasse?
- Come va dichiarata la rappresentazione di una classe astratta (con che modificatore di visibilità)? Perché?

Sidebar 7.4 Rep Invariant and Abstraction Function for Subclasses of Concrete Superclasses

- The abstraction function for a subclass, `AF_sub`, is typically defined using `AF_super`, the abstraction function of the superclass.
- The subclass rep invariant, `I_sub`, needs to include a check on the superclass rep invariant, `I_super`, only if the superclass has some protected members. However, `repok` for the subclass should always check `repok` for the superclass.

- Come si relazionano la funzione di astrazione e l'invariante di rappresentazione di una sottoclasse nei confronti della relativa superclasse?

Sidebar 7.5 Use of Protected Members

- It is desirable to avoid the use of protected members for two reasons: without them, the superclass can be reimplemented without affecting the implementation of any subclasses; and protected members are package visible, which means that other code in the package can interfere with the superclass implementation.
- Protected members are introduced to enable efficient implementations of subclasses. There can be protected instance variables, or the instance variables might be private, with access given via protected methods. The latter approach is worthwhile if it allows the superclass to maintain a meaningful invariant.

- Cosa comporta l'utilizzo del modificatore di visibilità **protected** (cosa succede quando lo si usa)?
- Quali rischi può comportare l'utilizzo di questo modificatore?

Sidebar 7.6 Reasoning about the Substitution Principle

- The *signature rule* ensures that if a program is type-correct based on the supertype specification, it is also type-correct with respect to the subtype specification.
- The *methods rule* ensures that reasoning about calls of supertype methods is valid even though the calls actually go to code that implements a subtype.
- The *properties rule* ensures that reasoning about properties of objects based on the supertype specification is still valid when objects belong to a subtype. The properties must be stated in the overview section of the supertype specification.

- Cosa afferma il **principio di sostituzione di Liskov**? Perché è importante che non venga infranto? Come si garantisce il principio di sostituzione?
- Cosa afferma la **regola delle segnature**? Chi ci assicura che venga rispettata?
- Cosa afferma la **regola dei metodi**? Chi ci assicura che venga rispettata?
- Cosa afferma la **regola delle proprietà**? Chi ci assicura che venga rispettata?
- Com'è in relazione una specificazione nei confronti del meccanismo di ereditarietà? In che senso, dato un tipo T e un sottotipo S, Le precondizioni di T implicano le precondizioni di S? In altri termini, cosa si intende per il **rilassamento delle pre-condizioni** dalla specificazione di un supertipo a quella del suo sottotipo? In che senso invece **vengono rafforzate le post-condizioni**? (Pensare ad esempi a riguardo e pensare ad esempi in cui si spiega cosa succede se queste regole non vengono rispettate)

Sidebar 7.8 Benefits of Hierarchy

- Hierarchy can be used to define the relationship among a group of types, making it easier to understand the group as a whole.
- Hierarchy allows code to be written in terms of a supertype, yet work for many types—all the subtypes of that supertype.
- Hierarchy provides extensibility: code can be written in terms of a supertype, yet continue to work when subtypes are defined later.
- All of these benefits can be obtained *only* if subtypes obey the substitution principle.

- Quali sono i benefici nell'utilizzare un meccanismo come quello delle gerarchia dei tipi?
- Cosa si intende dire che il codice dev'essere scritto tenendone sempre in mente l'estensibilità futura? In che modo il meccanismo di ereditarietà consente questo aspetto?

• CAPITOLO 8: POLYMORPHIC ABSTRACTION

Sidebar 8.1 Polymorphism

- Polymorphism generalizes abstractions so that they work for many types. It allows us to avoid having to redefine abstractions when we want to use them for more types; instead, a single abstraction becomes much more widely useful.
- A procedure or iterator can be polymorphic with respect to the types of one or more arguments. A data abstraction can be polymorphic with respect to the types of elements its objects contain.

- Cosa si intende per **polimorfismo**? Che diverse tipologie di polimorfismo esistono e cosa coinvolgono?
- Cosa si intende dire quando si parla di **variabili polimorfe**? Qual è l'importante conseguenza del fatto che in Java esista il polimorfismo sulle variabili (dunque una variabile non è associata unicamente ad uno e un solo tipo)?
- Cosa si intende quando si parla di **polimorfismo associato ai metodi**? Cosa significa dire che un metodo viene **sovrascritto**? Cosa significa dire che un metodo viene **sovraccaricato**? Fare un esempi a supporto delle spiegazioni di questi concetti
- Cosa si intende per **polimorfismo parametrico**? Tramite quali aspetti del linguaggio Java si concretizza questo aspetto del polimorfismo?
- Che relazione c'è tra i tipi generici e la relazione di sottotipo?

- # CAPITOLO 9: SPECIFICATIONS

Sidebar 9.1 Specifications

- A specification describes the behavior of some abstraction.
- An implementation *satisfies* a specification if it provides the described behavior.
- The meaning of a specification is the set of all programs that satisfy it. This set is called the *specificand set*.

- A cosa servono in generale le specificazioni?
- Cosa si intende per **insieme specificando**?

Sidebar 9.2 Attributes of Good Specifications

- A specification is *sufficiently restrictive* if it rules out all implementations that are unacceptable to an abstraction's users.
- A specification is *sufficiently general* if it does not preclude acceptable implementations.
- A specification should be *clear* so that it is easy for users to understand.

- Che caratteristiche deve avere una buona specificazione? Come deve essere in relazione al suo insieme specificando?
- Cosa comportano l'eccessiva restrittività e/o l'eccessiva generalità nella scrittura di una specificazione? Pensare ad esempi concreti

Sidebar 9.3 Benefits of Specifications

- An abstraction is intangible; without a description, it has no meaning. The specification provides this description.
- Writing a specification sheds light on the abstraction being defined, encouraging prompt attention to inconsistencies, incompleteness, and ambiguities. It forces us to pay careful attention to the abstraction and its intended use.
- A specification defines a contract between users and implementors: implementors agree to provide an implementation that satisfies the specification, and users agree to rely on not knowing which member of the specificand set is provided.